Mathematical Induction

- of no use for deriving formulas
- a good way to prove the validity of a formula that you might think is true

- never confuse MI with Inductive Attitude in Science. The latter is just a process of establishing general principles from particular cases.
- Statements proven by math induction all depend on an integer

Assume you want to prove that for some statement P, P(n) is true for all n starting with n = 1. The Principle (or Axiom) of Math Induction states that for this purpose you should accomplish just two steps:

1. Prove that P(1) is true.
2. Assume that P(k) is true for some k. Derive from here that P(k+1) is also true.

- Often it's impractical to start with n = 1. MI applies with any integer n_0. The result is then proven for all n starting with n_0.
- Sometimes, instead of 2., one assumes 2':
  Assume that P(m) is true for all m < (k+1).
  Derive from here that P(k+1) is also true.
  The two approaches are equivalent.

Suppose you want to prove a theorem in the form
"For all integers n greater than equal to a, P(n) is true"

- P(n) must be an assertion that we wish to be true for all n = a, a+1, ...; like a formula
- First, verify the **initial step** ® verify that P(a) is true
- **inductive step**
  - prove
    "If there is a k, greater than or equal to a, for which P(k) is true,
    then for this same k, P(k+1) is true."

Big-Oh notation

- To establish a relative order among function
- Compare relative rates of growth

Big-Oh notation

- efficiency indicator
- instead of saying "order…," one says "Big-Oh…"
- is used to compare two functions
- Typically, one function, which we'll call T(*N*), models the cost of an algorithm using some cost metric. The other, which we'll call f(*N*), gives an upper-bound on T(*N*).
- Interestingly, O(f(*N*)) is a *set* of functions, since f(*N*) is typically an upper bound on many different functions.
  - This is why we have said ``T(*N*) is *in* O(f(*N*))" rather than ``T(*N*) is O(f(*N*))".
- want to ignore constant multipliers
- Want to include the ``for sufficiently large *N*" in our formal definition, since we've seen that the higher-order terms only dominate when *N* is large enough.
- describes an *upper* bound on the running time of an algorithm.
- However, this bound need not be tight, nor does it tell us about lower bounds.

Definition

- T(N) = O(f(N)) if
  there are positive constants c and n_0 such that
  $T(N) \le c\, f(N)$ when $N \ge n_0$.

Note on the definition

- The *c* in this definition is used to account for constant multipliers.
  - Since we can choose the multiplier for f(*N*), it doesn't matter what multipliers we have for T(*N*), we can just choose a bigger one.
- The n_0 in this definition is used for ``sufficiently large".
  - We choose an n_0 large enough that the dominating term actually dominates.

Interpretation

- eventually, there is some point n_0 past which c f(N) is always at least as large as T(N)
- the growth rate of T(N) is ≤ to that of f(N)
- f(N) is an upper bound on T(N)
- f(N) = Ω(T(N)) → T(N) is a lower bound on f(N)

Big-O allows us to ignore constant multipliers

⇒

if f(N) is in O(C*g(N)),
then f(N) is in O(g(N)) for all positive C

proof

1. If f(N) is in O(C*g(N)), then there exist M_1 and $D_1 > 0$ such that for all $N > M_1$, $|f(N)| \le |D_1 * C * g(N)|$. [By the definition of Big-O.]
2. Let $M_2 = M_1$.
3. Let $D_2 = |D_1 * C|$.
4. For all $N > M_2$, $|f(N)| \le |D_1 * C * g(N)|$. [By step 1 and definition of M_2.]
5. For all $N > M_2$, $|f(N)| \le |D_2 * g(N)|$. [By definition of D_2.]
6. f(N) is in O(g(N)). [By definition of Big-O.]

Big-O allows us to ignore lower order terms

⇒

if

f(N) is in O(g(N) + h(N)) and
g(N) is in O(h(N))

then

f(N) is in O(h(N)).

proof

1. If f(N) is in O(g(N) + h(N)), then there exist M_1 and $D_1 > 0$ such that for all N greater than M_1, $|f(N)| \le |D_1 *(g(N) + h(N))|$. [By the definition of Big-O.]
2. If g(N) is in O(h(N)), then there exist M_2 and $D_2 > 0$ such that for all N greater than M_2, $|g(N)| \le |D_2 * h(N)|$.
3. Let $M_3 = \max(M_1, M_2)$. ⇒ $N > M_3$ is $\{N > M_1$ and $N > M_2\}$

For all $N > M_3$,

4. $|f(N)| \le |D_1 *(g(N) + h(N))|$. [By step 1 and definition of M_3.]
5. $|f(N)| \le D_1 * |g(N)+h(N)|$. [If D>0, then $|D*X| = |D|*|X|$.]
6. $|f(N)| \le D_1 * |g(N)| + D_1 * |h(N)|$. [$|X+Y| \le |X|*|Y|$.]
7. $|f(N)| \le D_1 * |D_2 * h(N)| + D_1 * |h(N)|$. [By step 2 and definition of M_3.]
8. $|f(N)| \le |(D_1 * D_2 + D_1) * h(N)|$. [Various arithmetical manipulations; $D_1 > 0$.]
9. Let $D_3 = D_1 * (1 + D_2)$.
10. $|f(N)| \le |D_3 * h(N)|$. [Definition of D_3.]
11. f(N) is in O(h(N)). [Definition of Big-O.]

Big-O is transitive

if

f(N) is in O(g(N)) and
g(N) is in O(h(N))

then f(N) is in O(h(N))

proof

1. Because f(N) is in O(g(N)), there exist M_1 and D_1 such that for all $N > M_1$, $|f(N)| \leq |D_1 * g(N)|$.
2. Because g(N) is in O(h(N)), there exist M_2 and D_2 such that for all $N > M_2$, $|g(N)| \leq |D_2 * h(N)|$.
3. Let $M_3 = \max(M_1, M_2)$.

Then for all $N > M_3$,

4. $|f(N)| \leq |D_1 * g(N)|$.
   [By the first rule and the definition of M_3.]
5. and
   $|g(N)| \leq |D_2 * h(N)|$.
   [By the second rule and the definition of M_3.]
6. Then
   $|D_1 * g(N)| \leq |D_1 * D_2 * h(N)|$.
   [Since $D_1 > 0$, we can safely multiply both sides of an inequality for D_1 without affecting the inequality.]
7. $|f(N)| \leq |D_1 * D_2 * h(N)|$.
   [We can plug together various inequalities using transitivity of inequality.]
8. Let $D_3 = \max(1, D_1) * \max(1, D_2)$.
9. $|f(N)| \leq |D_3 * h(N)|$.
   [$D_3 \geq D_1 * D_2$.]
10. Hence f(N) is in O(h(N)).
    [By the definition of Big-O.]

Rule

- If
  $T_1(N) = O(f(N))$ and
  $T_2(N) = O(g(N))$,
  then
  - $T_1(N) + T_2(N) = \max(O(f(N)), O(g(N)))$
  - $T_1(N) * T_2(N) = O(f(N) * g(N))$
- $\log^k N = O(N)$ for any constant k.
  $\Rightarrow$ logarithms grow very slowly

Include negative N

- to incorporate possibly negative results, we'll use absolute value when we formalize Big-O.
- T(N) is an element of O(f(N)) if and only if there exist constants c and n_0 such that for all $N > n_0$,
  $|T(N)| \leq |c * f(N)|$.

The function g(N) is a ***tight upper bound*** on f(N) if

1. f(N) is in O(g(N)) and
2. for all h(N) such that
   f(N) is in O(h(N)),
   g(N) is also in O(h(N)).

style:

- Don't include constants or low-order terms inside a Big-Oh
- Bad to say $f(N) \leq O(g(N))$
  because the inequality is implied by the definition
- Wrong to write $f(N) \geq O(g(N))$
  $\rightarrow$ does not make sense

- The running time of a **for** loop is at most
  the running time of the statements inside the for loop (including tests)
  times
  number of iterations.
- Nested loop $\rightarrow$ Analyze these inside out.
- Consecutive statement $\rightarrow$ just add $\rightarrow$ the maximum is the one that counts
- if(condition)
  S1
  else
  S2
  the running time of an if/else statement is never more than
  the running time of the test +
  the larger of the running times of S1 and S2
  - can be an overestimate, but never an underestimate

- An algorithm is O(log N) if
  it takes constant (O(1)) times to cut the problem size by a fraction (which is usually $\frac{1}{2}$)
- An algorithm is O(N) if
  constant time is required to reduce the problem by a constant amount
- $O(\log_a N) = O(\log N)$
- all the $\log_a(N)$ are big-O of each others.

loop → Σ

- for(int i = 0; i<n; i++)
  some operation that is O(1)
  $$\to \sum_{i=0}^{n-1} O(1) = n \cdot O(1) = O(n)$$
- for(int i = 0; i<n; i++)
  some operation that is $O(n^k)$
  $$\to \sum_{i=0}^{n-1} O(i^k) = O\left(\sum_{i=0}^{n-1} i^k\right) = O(n^{k+1})$$
- for(int i = 0; i<g(n); i++)
  some operation that is $O(n^k)$
  $$\to \sum_{i=0}^{g(n)} O(i^k) = O\left(\sum_{i=0}^{g(n)} i^k\right) = O\left((g(n))^{k+1}\right)$$
- $$\sum_{i=0}^{O(n)} O(x^i) = O(x^n)$$

To analyze recursion

- assume the running time is T(N)
- write out the equation from the recursion relation from the definiton of the method in terms of T(k) where k < N, constant, and function of N

- a commonly used trick to verify that some program is O(f(N)):
  T(N) = empirically observed running time
  compute the value $\frac{T(N)}{f(N)}$ for a range of N
  (usually spaced out by factors of 2)
  - o converge to a positive constant → tight answer
  - o converge to 0 → overestimate
  - o converge to ∞ → underestimate → wrong

Typical growth rates

Function	Name	
1		• retrieval of a single data item from an array, when you know where the data item is
c	constant	
log N	logarithmic	• the NCAA basketball tournament • average and worse-case efficiency of a binary search
$\log^2 N$	log-squared	
N	linear	• average and worse-case efficiency of performing a sequential search for a data item in an array
N log N		• average efficiency of performing a mergesort or a quicksort on an array • space require for writing n number (each number take log N length text to denote it value) • space require for keeping N data in a list (take log N to store the link that differentiate all N data)
N^2	quadratic	• average and worse-case efficiency of bubblesort or selection sort • worse-case efficiency of a quicksort
N^3	cubic	
2^N	exponential	• Tower of Hanois
n!		• combinatoric problems

Other definition

$$\lim_{N\to\infty} \frac{f(N)}{g(N)} = \begin{cases} 0 & \Rightarrow \quad f(N) = o(g(N)) \\ c \neq 0 & \Rightarrow \quad f(N) = \Theta(g(N)) \\ \infty & \Rightarrow \quad g(N) = O(f(N)) \\ \text{oscillate} & \Rightarrow \quad \text{no relation} \end{cases}$$

- **Omega**: T(N) = **W**(g(N)) if
  there are positive constants c and n_0 such that
  $T(N) \geq c\, g(N)$ when $N \geq n_0$.
  ⇒ the growth rate of T(N) is ≥ that of g(N)
- **Theta**: T(N) = **Q**(h(N)) if and only if
  T(N) = O(h(N)) and
  T(N) = Ω(h(N))
  ⇒ the growth rate of T(N) is = that of h(N)
  - If T(N) is a polynomial of degree k, then $T(N) = \Theta(N^k)$
- **Little-oh**: T(N) = **o**(p(N)) if
  T(N) = O(p(N)) and
  T(N) ≠ Θ(p(N))
  ⇒ the growth rate of T(N) is < that of p(N)

Recursion

- a function that is defined in terms of itself is called recursive
- **base case** → the value for which the function is directly known without resorting to recursion
- should never be used as a substitute for a simple for loop

Thing to keep in mind

- Base case → Must always have some base cases, which can be solved without recursion
- Making progress → For the case that are to be solved recursively, the recursive call must always be to a case that makes progress toward a base case
- Design rule → Assume that all the recursive calls work
- Compound interest rule → Never duplicate work by solving the same instance of a problem in a separate recursive call
  - Don't compute anything more than once

Proof by Induction	Recursion
• Define some measure of the problem "N" • BASE CASE: Prove the theorem holds for some base case (N=0 or N=1) • EXTENSION CASE: Assume we have proven the theorem for the "K-th" case, show the proof holds for the "K+1-st" case. (the immediate problem) • Since we want a proof for any N, you can stop here (at N=K, since K is any value anyway).	• Define some measure of the problem "N" • BASE CASE: Solve the problem for a base case • EXTENSION CASE: Assume you can solve the problem for the "K-th" case by calling yourself recursively for problem of size "K", using solution returned, solve the problem for the "K+1-st" case in this immediate step. • Stop when K=N or (if working backwards from N, when K=the base case).

- **tail recursion** → a recursive call at the last line
- can be machanically eliminated by enclosing the body in a while loop and replacing the recursive call with an assignment per method argument (some compilers do it automatically)

- recursion can always be completely removed
- compilers do so in converting to assembly language

although nonrecursive programs are certainly generally faster than recursive programs, the speed advantage rarely justifies the lack of clarity that results from removing the recursion.

Abstract Data Types (ADTs)

⇒ a set of objects together with a set of operations

List

Simple Array Implementation of lists

- insertion/deletion → O(N)
- building a list by successive insert → $O(N^2)$

Linked lists

- space require for keeping N links to N data $\rightarrow$ O(N log N)
  size of each address is O(log N) $\rightarrow$ to differentiate all N data
- access i^{th} object, starting from head $\rightarrow$ O(i)
  access all object in the list $\rightarrow O(n^2)$
- keep track of the last element $\rightarrow$ access all object in the list $\rightarrow$O(n)

Iterator

- using a separate iterator class express the abstraction that the position and list are really separate objects.
- allows for a list to be accessed in several places simultaneously

```
class DataHolder //node
{
   Object ob;
   DataHolder next;
   DataHolder(Object ob)
   {
      //choose whether to make a copy
      //or use the reference
      this.ob = ob;
      next = null;
   }
}
```

```
public class LinkedList
{
   DataHolder first,last;
   LinkedList()
   {
      first = null;
      last = null;
   }
   void addToEnd(Object ob)
   {
      DataHolder dh = new DataHolder(ob);
      //the list is empty
      if(first==null)
      {
         first = last = dh;
      }
      //this list is not empty
      else
      {
         last.next = dh;
         last = dh;
      }
   }
```

```
   void insert(Object ob, int place)
   {
      //insert to be the first element
      if(place==0)
      {
         DataHolder dh = new DataHolder(ob);
         dh.next = first;
         first = dh;
      }
      else
      {
         //think of inserting at n as
         //inserting at after n-1
         DataHolder precede = at(place-1);
                  //position n-1
         if(precede==null){/*error*/}
         DataHolder dh = new DataHolder(ob);
         dh.next = precede.next;
         precede.next = dh;
      }
   }
```

```
    DataHolder at(int n)
    {
       DataHolder finger = first;
       if(first==null) return null; //empty list
       while(n>0)
       {
          //too big n
          if(finger.next == null) return null;
          finger = finger.next;
          n--;
       }
       return finger;
    }
    Object objectAt(int n)
    {
       DataHolder dh = at(n);
       if(dh==null) return null;
       return dh.ob;
    }
```

```
    void traverse() //O(n^2)
    {
       int i = 0;
       String s = (String)objectAt(i);
       while(s!=null)
       {
          System.out.print(s + " ");
          i++;
          s =  (String)objectAt(i); //O(n)
       }
       System.out.print("\n");
    }
```

```
    void traverseF() //O(n)
    {
       DataHolder finger = first;
       while(finger!=null)
       {
          System.out.print((String)finger.ob + " ");
          finger=finger.next;
       } //violate encapsulation
       System.out.print("\n");
    }
```

```
   void traverseI() //O(n)
   {
      LLIterator li = new LLIterator(this);
      while(li.hasNext())
      {
         System.out.print((String)li.getNext().ob+ "
");
      }
      System.out.print("\n");
   }
}
```

```
class LLIterator
{
   //the one you will return
   //when using the getNext call
   DataHolder current;
   LLIterator(LinkedList LL)
   {
      current = LL.first;
   }
   boolean hasNext()
   {
      return(current!=null);
   }
   DataHolder getNext()
   {
      DataHolder ans = current;
      if(current!=null)
         current=current.next;
      return ans;
   }
}
```

Other

- Doubly linked list
- Circular linked list

Stack

- `LIFO` (last in, first out) lists
- *inserts* and *deletes* can be performed in only one position, namely the end of the list called the *top*.
- The fundamental operations on a stack are
    - *push* ⇒ insert, and
    - *pop* ⇒ deletes the most recently inserted element.
    - The most recently inserted element can be examined prior to performing a *pop* by use of the *top* routine.
- A *pop* or *top* on an empty stack is generally considered an error in the stack `ADT`.
- running out of space when performing a *push* is an implementation error but not an `ADT` error.
- both the linked list and array implementations gives fast O(1) running times for every operation

Balancing parentheses

- For simplicity, we will just check for balancing of parentheses, brackets, and braces and ignore any other character that appears.

- Make an empty stack.
- Read characters until end of file.
- If the character is an open anything, push it onto the stack.
- If it is a close anything, then
    - if the stack is empty report an error.
    - Otherwise, pop the stack.
        - If the symbol popped is not the corresponding opening symbol, then report an error.
- At end of file, if the stack is not empty report an error.

Postfix Expressions

- *postfix* or *reverse Polish* notation
- there is no need to know any precedence rules

- When a number is seen, it is pushed onto the stack;
- when an operator is seen, the operator is applied to the two numbers (symbols) that are popped from the stack and the result is pushed onto the stack.
- The time to evaluate a postfix expression is $O(n)$

Infix to Postfix Conversion

- concentrate on a small version of the general problem by allowing only the operators +, *, and (,), and insisting on the usual precedence rules.
- assume that the expression is legal

- start with an initially empty stack.
- When an operand is read, it is immediately placed onto the output.
- place operators that have been seen onto the stack
- stack left parentheses when they are encountered.
- If we see a right parenthesis, then we pop the stack, writing symbols until we encounter a (corresponding) left parenthesis, which is popped but not output.
- If we see any other symbol ('+','*', '('), then we pop entries from the stack **until we find an entry of lower priority**.
    - Exception: never remove a '(' from the stack except when processing a ')'
    - When the popping is done, we **push the symbol onto the stack**.
- Finally, if we read the end of input, we pop the stack until it is empty, writing symbols onto the output.

Queue

- is list
- insertion is done at one end, whereas deletion is performed at the other end
- both the linked list and array implementations gives fast O(1) running times for every operation

basic operations on a queue are

- *enqueue* ⇒ inserts an element at the end of the list (called the rear),
- *dequeue* ⇒ deletes (and returns) the element at the start of the list (known as the front).

Tree

- is a collection of nodes
- consists of
    - a distinguished node $r \rightarrow$ the *root*, and
    - zero or more (sub)trees $T_1, T_2, \ldots, T_k$, each of whose roots are connected by a directed *edge* to r
- The collection can be empty, which is sometimes denoted as A
- The root of each subtree $\rightarrow$ a *child* of r, and $r \rightarrow$ *parent* of each subtree root.
- Each node may have an arbitrary number of children, possibly zero.
- Nodes with no children $\rightarrow$ *leaves*
- Nodes with the same parent $\rightarrow$ *siblings*
- *grandparent*
- *grandchild*
- A *path* from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, \ldots, n_k$ such that n_i is the parent of n_{i+1} for $1 \leq i < k$.
- The *length* of this path is the number of edges on the path, namely k -1.
- There is a path of length zero from every node to itself.
- in a tree there is exactly one path from the root to each node
- the **depth** of node n_i is the length of the unique path from the root to n_i.
- the root is at depth 0.
- The *height* of node n_i is the longest path from n_i to a leaf.
- all leaves are at height 0.
- The height of a tree = the height of the root.
- If there is a path from n_1 to n_2, then n_1 is an *ancestor* of n_2 and n_2 is a *descendant* of n_1.
    - If $n_1 \neq n_2$, then
        - n_1 is a *proper ancestor* of n_2
        - n_2 is a *proper descendant* of n_1

- running time of most operations is O(log n) on average
- a tree is a collection of n nodes, one of which is the root, and n - 1 edges.
    - That there are n - 1 edges follows from the fact that each edge connects some node to its parent, and every node except the root has one parent
- generally drawn as circles connected by lines (because they are actually graph)
- do no explicitly draw null links

- One natural way to define a tree is recursively

Implementation

- to have in each node, besides its data, a pointer to each child of the node.
- since the number of children per node can vary so greatly and is not known in advance, it might be infeasible to make the children direct links in the data structure, because there would be too much wasted space.
- solution $\rightarrow$ keep the children of each node in a linked list of tree nodes.

```
class TreeNode
{
   Object element;
   TreeNode firstChild;
   TreeNode nextSibling;
}
```

Binary Trees

- no node can have more than 2 children
- N nodes have N+1 null links
  2N – N + 1(root)
-

Expression trees

- leaves nodes contain operands
- other nodes contain operators.
- To evaluate $\rightarrow$ applying the operator at the root to the values obtained by recursively evaluating the left and right subtrees
- inorder traversal (left node right)
- postorder traversal (left right node)
- preorder traversal (node left right)
- constructing an expression tree from postfix string
    - o read expression 1 symbol at a time
    - o if symbol is an operand $\rightarrow$ create one-node tree and push it onto stack
    - o if symbol is an operator $\rightarrow$ pop 2 trees T_1 and T_2 from the stack (T_1 is popped first) and form a new tree whose root is the operator and whose left and right children are T_2 and T_1. This new tree is then pushed onto the stack.

```
class BinaryTree
{
   String data;
   BinaryTree left,right;
   BinaryTree(String d, BinaryTree l, BinaryTree r)
   {
      data = d;
      left = l;
      right = r;
   }
   BinaryTree(String d)
   {
      this(d,null,null);
   }
}
```

```
class ArTree extends BinaryTree
{
   int evaluate()
   {
      if(left==null && right==null)
         return Integer.parseInt(data);
      if(data.equals("+")
         return left.evaluate()+right.evaluate();
      if(data.equals("*")
         return left.evaluate()*right.evaluate();
   }
```

```
   String infixString()
   {
      if(left==null && right==null)
      {
         return data;
      }
      return "(" + left.infixString() + data +
                right.infixString() + ")";
   }
```

```
   String postfixString()
   {
      if(left==null && right==null)
      {
         return data;
      }
      return  left.postfixString() +
                right.postfixString() + data;
   }
}
```

constructing an expression tree from infix string

- o Depth of an operator in terms of parens can be found by using stack, depth is the number of "(" 's that are found in the stack when encounter the operator.

```
{
   Boolean isANumber();
   Boolean isParenthesized();
   Boolean isCompound();
   void deParenthesized();
   //if there is a +,- otherwise *,/ at level 0
   //(surrounded by 0 parens)
   //take the first(last?) one
   String getOperator();
   String getLeftExp();
   String getRightExp();
   void buildTree()
   {
      if(isParenthesized())
         deParenthesized();
      if(isANumber())
         return;
      if(isCompound())
      {
         String op = getOperator();
         String leftString = getLeftExp();
         String rightString = getRightExp();
         data = op;
         left = new BinaryTree(leftString);
         right = new BinaryTree(rightString);
         left.buildTree();
         right.buildTree();
      }
   }
}
```

Tree traversal

- preorder: node, children(in order)
- inorder: 1st child, node, other children
- postorder: children(in order), node

Analyzing game using tree

- tree may log the same state multiple times

Consider the game:

- Set value for each node with respect to player x
- win = 1, draw = 0, lose = -1
- If work by hand → go from deepest level up (the value at leaves is known)
- If work recursively →
  - at leaves (the end of the game), return the value of the node
  - at node followed by the edge of player X's move → return the max value of children nodes

    → equivalent to choosing the best move for player X
  - at node followed by the edge the is not of player X's move → return the min value of children nodes

    → equivalent to choosing the worst move for player X

- Graph
  - can have multiple pathways to a state
  - if can afford to ignore history, we can make a smaller graph by only recording state and transition.

Binary search tree

- depends on
  - data and
  - order of insertion
- worst case for the depth is when the data are already sorted.
- average depth: O(log n)
- worst case building: $O(n^2)$
- average building: $O(n) \times O(d) = O(n \log n)$

Properties

- for every node, *X*, in the tree,
  - the values of all the items in the left sub-tree are smaller than the item in *X*, and
  - the values of all the items in the right sub-tree are larger than the item in *X*.

  → make a binary tree into a binary search tree

```
class BinaryTree
{
   Comparable data;
   BinaryTree left,right;
   BinaryTree(String d, BinaryTree l, BinaryTree r)
   {
      data = d;
      left = l;
      right = r;
   }
   BinaryTree(Comparable d)
   {
      this(d,null,null);
   }
   BinaryTree()
   {
      this(null,null,null);
   }
```

Insertion

- if at an empty node → insert
- otherwise recursively insert into
  - o left sub-tree if less than current value
  - o right sub-tree if greater than current value
- worst case : O(n^2)

```
   void insert(Comparable x)
   {
      if(data==null)
      {
         data = x;
      }
      else if(x.compareTo(data)<0)
      {
         if(left==null) left=new BinaryTree(x);
         else left.insert(x);
      }
      else if(x.compareTo(t.data) > 0)
      {
         if(right==null) right=new BinaryTree(x);
         else right.insert(x);
      }
      else
         doSomething(); //Match
   }
```

Find

```
   Comparable find(Comparable x)
   {
      if(x.compareTo(data)<0)
      {
         if(left==null) return null;
         return left.find(x);
      }
      else if(x.compareTo(t.data) > 0)
      {
         if(right==null) return null;
         return right.find(x);
      }
      else
         return data; //Match
   }
```

- crutial that the test for an empty tree be performed first, since otherwise, we would generate a NullPointerException attempting to access a data field through a null reference.
- both recursive calls are actually tail recursions
  - o can be easily removed with a while loop.

```
   Comparable findMin()
   {
      if(left==null) return data;
      return left.findMin();
   }
   Comparable findMax()
   {
      BinaryTree temp = this;
      while(right!=null)
      {
         temp = temp.right;
      }
      return temp.data;
   }
```

Deletion

- a leaf $\longrightarrow$ just delete it
- a node with a single child $\longrightarrow$ can be deleted after its parent adjusts a link to bypass the node
- a node with 2 children $\longrightarrow$ replace the data of this node with the smallest data of the right subtree (which is easily found) and recursively delete that node.

```
   void delete(Comparable x)
   {
      if(x.compareTo(data)<0)
      {
         if(left==null)
         {
            //no x in this tree
            return;
         }
         else left.delete(x);
      }
      else if(x.compareTo(data)>0)
      {
         if(right==null)
         {
            //no x in this tree
            return;
         }
         else left.delete(x);
      }
      else
      {
         //x is in this node
         if(left==null && right==null)
         //this is a leaf, just delete it
         {
            this = null;
         }
         // Nodes that has only one child
         //(left or right)
         else if(left!=null && right==null)
         {
            this = this.left;
         }
         else if(left==null && right!=null)
         {
            this = this.right;
         }
         else //have left and right children
         {
            data = right.findMin();
            right.delete(data);
            //this node cannot have left child
            //so it won't fall into this case again
         }
      }
      return;
   }
}
```

- To get the data out in sorted order $\longrightarrow$ traverse in inorder
- The running time of all the operations is O(depth)
  depth = depth of the node containing the accessed item
- after O(n^2) #insertions=#deletions , depth is $O\left(\sqrt{n}\right)$

independent method for binary search tree

```
public static int binarySearch( Comparable [a],
                                Comparable x)
{
   int low = 0;
   int high = a.length-1;

   while(low <= high)
   {
      int mid = (low+high) / 2;
      //if the middle is less than x,
      //change to consider only the right half
      if( a[mid].compareTo(x) < 0 )
         low = mid+1;
      //if the middle is more than x,
      //change to consider only the left half
      else if( a[mid].compareTo(x) > 0 )
         high = mid-1;
      else
         return mid; //found
   }
   return NOT_FOUND;
}
```

AVL Trees

(Adelson-Velskii and Landis)

- a binary search tree with a balance condition
- depth of the tree is O(log N)
- for every node in the tree, the height of the left and right subtrees can differ by at most 1
- the height of an empty tree is defined to be -1

Binary Heap

⇒ a binary tree that is completely filled, with the possible exception of the bottom level, which is filled from left to right

- implement Priority Queues (Heaps)
- heap-order property → any node is smaller than all of its descendants
- the minimum element can always be found at the root findMin() → O(1)

complete binary tree

- the height of a complete binary tree is $\lfloor \log N \rfloor$ = O(log N)
    - o can be represented in an array and no links are necessary
    - o no element in array[0]
    - o for any element in array position i
        - left child = array[2*i]
        - right child = array[2*i+1]
        - parent = array$\left[\left\lfloor \frac{i}{2} \right\rfloor\right]$
    - o estimate of the maximum heap size is required in advance

insert x

<u>percolate up</u>

- create a hole in the next available location
- if x can be placed in the hole without violating heap order, then we do so and are done.
- otherwise, slide the element that is in the hold's parent node into the hole, thus bubbling the hole up toward the root
- continue this process until x can be placed in the hole

```
void insert(Comparable x) //O(log N)
{
   int hole = ++currentSize();
   for(;hole > 1 &&x.compareTo(array[hole/2])<0;
                                hole /= 2)
   {
      //get value from its parent
      array[hole]=array[hole/2];
   }
   array[hole] = x;
}
```

- sentinel: small value in position 0 inorder to make the loop terminate
    - o must be guaranteed to be ≤ any element in the heap

deleteMin

- create a hole at the root
- store the last element X
- if X can be placed in the hole, done
- otherwise, slide the smaller of the children into the hole, thus pushing the hole down one level
- repeat until X can be placed in the hole

```
void deleteMin() //O(log N)
{
   int hole = 1; //the root

   array[hole] = array[ currentSize-- ];

   //percolate down

   int child;
   Comparable temp = array[hole];
   for(;hole*2 <= currentSize; hole=child)
   {
      child=hole*2; //left child
      //if right child is less than left child,
      //use right child
      if(child!=currentSize &&
         array[child+1].compareTo(array[child])<0)
      {
         child++;
      }
      if(array[child].compareTo(tmp)<0)
      {
         array[hole]=array[child]; //move child up
      }
      else
         break;
   }
   array[hole] = temp;
}
```

- can buildHeap() with N successive inserts → O(log N)*O(N) = O(N log N)
- finding the kth smallest element: perform k deleteMin operation → O(buildHeap) + O(k log N)
-

Graph

G = (V,E)

- V = set of vertices
- E = set of edges
- $|E| = O(|V|^2)$ (can be a lot less than this)
- **edge** = arcs = a pair (v,w) where v,w ∈ V
  - if the pair is ordered, the graph is directed → **digraph**
  - optional component: weight/cost
- vertex w is **adjacent** to v if and only if (v,w) ∈ E
- **path** = sequence of vertices $w_1,\ldots,w_N$ such that $(w_i,w_{i+1}) \in E$ for $1 \le i < N$.
  - o **length** = number of edges on the path = N-1
  - o allow path from vertex to itself
    - ▪ if this path contains no edges, the path length = 0
  - o **simple path** → all vertices are distinct, except that the first and the last could be the same
- **cycle**
  - o in a directed graph → path of length at least 1 such that $w_1 = w_M$
  - o in undirected graph → distinct edges
- **DAG**: **directed acyclic graph** → directed graph with no cycle

- **connected** undirected graph $\longrightarrow$ there is a path from every vertex to every other vertex
- **strongly connected directed** graph $\longrightarrow$ there is a path from every vertex to every other vertex
- **weakly connected directed** graph $\longrightarrow$ the underlying graph (without direction to the arcs) is connected
- **complete graph** $\longrightarrow$ there is an edge between every pair of vertices
- **indegree** of a vertex v $\longrightarrow$ #edges (u,v)
- **weighted graph** $\longrightarrow$ associated with each edge (v_i,v_j) is a cost $c_{i,j}$ to traverse the edge
- **weigthed path length** $\longrightarrow$ the cost of a path $v_1 \ldots v_N = \sum_{i=1}^{N-1} c_{i,i+1}$
- **unweighted path length** $\longrightarrow$ the number of edges on the path = N-1
- **diameter** $= \max_{v,w\in V}\{dist(v,w)\}$

Representation of a graph

- **adjacency matrix** $\longrightarrow$ use 2-d array
  - space: $O(|V|^2)$
  - appropriate if the graph is ***dense***: $|E| = \Theta(|V|^2)$
- **adjacency list**
  - the standard way to represent graphs
  - space: $O(|E|+|V|)$

Topological sort

$\Rightarrow$ an ordering of vertices in a directed acyclic graph, such that if there is a path from v_i to v_j, then v_j appears after v_i in the ordering

- not possible if the graph has a cycle

- find any vertex with no incoming edges, print it and remove it along with its edge
- apply this same strategy to the rest of the graph

```
void topsort() throws CycleFound //O(|V|^2)
{
   Vertex v, w;

   for(int counter=0; counter<|V|; counter++)
   {
      //look for a vertex with indegree 0
      //that has already been assigned topNum
      v = findNewVertexOfDegreeZero(); //O(|V|)
      if(v=null)
         throw new CycleFound();
      v.topNum = counter;
      for each w adjacent to v
         w.indegree--;
   }
}
```

unweighted shortest-path

Given an unweighted graph G. Using some vertex, s, which is am input parameter, find the shortest path from s to all other vertices

- consider only the #edges contained on the path
- no weights on the edge
- special case of weighted shortest-path problem, could assign all edges a weight of 1.
- breadth-first search

- look for all vertices that are at distance 1 away from s $\longrightarrow$ vertices that are adjacent to s
-

```
void reset()
{
   currMaxDist = 0;
   for(int i=0;i<vertex.size();i++)
   {
      Vertex v = vertex.At(i);
      v.known = false;
      v.dist = INF;
      v.via = null;
   }
}
```

```
void AssignDist(int base) //O(|V|^2)
{
    Vertex v, w;
    vertex.At(base).dist = 0;
    for(int currDist=0; currDist < vertex.size(); currDist++)
    {
        for(int i = 0; i<vertex.size();i++)
        {
            v = vertex.At(i);
            if(!v.known && v.dist==currDist)
            {
                v.known = true;
                List adjList = edge.elementAt(i);
                //for each w adjacent to v
                for(int j =0; j<adjList.size();j++)
                {
                    w = adjList.At(j);
                    if(w.dist==Vertex.INF)
                    {
                        w.dist = currDist+1;
                        w.path = v;
                        currMaxDist=currDist+1;
                    }
                }
            }
        }
    }
}
```

```
void AssignDist(int base)
{
    Vertex v, w;
    vertex.At(base).dist = 0;
    vertex.At(base).known = true;
    boolean allKnown = false;
    while(!allKnown)
    {
        allKnown = true;
        for(int i = 0; i<vertex.size();i++)
        {
            v = vertex.At(i);
            if(v.known)
            {
                List adjList = edge.At(i);
                for(int j =0; j<adjList.size();j++)
                {
                    w = adjList.At(j);
                    if(!w.known)
                    {
                        w.dist = v.dist+1;
                        w.known = true;
                        currMaxDist=w.dist;
                    }
                }
            }
            else{allKnown = false;}
        }
    }
}
```

single-source shortest path problem

Given as input a weighted graph, G=(V,E), and a distinguished vertex, s, find the shortest weighted path from s to every other vertex in G

- no edge of negative cost → the shortest path between two points is undefned
- negative cost cycle → when one is present in the graph, the shortest path are not defined
- the shortest path from s to s is 0
- currently, there are no algorithms in which finding the path from s to one vertex is any faster (by more than a constant factor) than finding the path from s to all vertices

Initialization

- all vertex.known = false
- all vertex.via = null
- all vertex.dist = ∞

set basePoint.dist = 0;

Dijkstra's algorithm

- solve the single-source shortest-path problem
- for weighted graph
- for unweighted graph, assign all weight to be 1
- works with directed or undirected graph
- greedy algorithm → generally solve a problem in stages by doing what appears to be the best thing at each stage
    - do not always work
- d_v = the shortest path length from s to v using only known vertices as intermediates
- p_v = the last vertex to cause a change to d_v

v	known	dist	p_v

```
while(not all known)
{
    find unknown with min dist v
    mark it known
    find all adjacent vertices
    for each unknown adjacent vertices w
    {
        if(v.dist + weight of edge (v,w) < w.dist )
        {
            update dis w
            set w.via = v
        }
    }
}
```

```
void dijkstra(int base)
{
    reset();
    Vertex v,w;
    vertex.At(base).dist = 0;
    for(;;)
    {
        //v = the smallest unknown distance vertex
        int minUnknownV = noV;
            //this will get changed
            //if unknown vertex(s) is found
        int minUnknownValue = Vertex.INF;
        for(int i=0;i<vertex.size();i++)
        {
            Vertex p = vertex.At(i);
            if(!p.known)
            {
                if(p.dist < minUnknownValue )
                {
                    minUnknownValue = p.dist;
                    minUnknownV = i;
                }
            }
        }
        //minUnknownV is still nopath
        //→ no unknown vertex → done
        if(minUnknownV == noV)
            break;

        v = vertex.At(minUnknownV);
        v.known = true;
        List adjList = edge.At(minUnknownV);
        //for each unknown w adjacent to v
        for(int i=0; i<adjList.size(); i++)
        {
            w = adjList.At(i));
            if(!w.known)
            {
                int d = distance(v,w);
                if(v.dist + d < w.dist)
                {
                    w.dist = v.dist + d;
                    w.via = v;
                }
            }
        }
    }
}
```

```
void printPath(Vertex v)
{
    if(v.path != null)
    {
        printPath(v.path);
        System.out.print(" to ");
    }
    System.out.print( v );
}
```

- always works as long as no edge has a negative cost
- running time
  - if scanning down array to find minimum d_v
    each stage use O(|V|)
    have to run this O(|V|)
    so, use $O(|V|^2)$ to find minimum in this code
    the update is O(|E|)
    so running time: $O(|E|+|V|^2) = O(|V|^2)$
  - to date, there are no meaningful average-case results for this problem

Example

$0 \rightarrow 1(10), 2(1)$
$1 \rightarrow 0(10), 3(1)$
$2 \rightarrow 0(1), 3(1)$
$3 \rightarrow 2(1), 1(1)$

0	0	F	null
1	∞	F	null
2	∞	F	null
3	∞	F	null

•unknown with min dist = 0

0	0	T	null
1	∞	F	null
2	∞	F	null
3	∞	F	null

•0 is connected to 1,2 both unknown

•for 1: $0+10 < \infty$

0	0	T	null
1	10	F	0
2	∞	F	null
3	∞	F	null

•for 2: $0+1 < \infty$

0	0	T	null
1	10	F	0
2	1	F	0
3	∞	F	null

not all known

- unknown with min dist = 2

0	0	T	null
1	10	F	0
2	1	T	0
3	∞	F	null

- 2 is connected to 0,3 only 3 is unknown
- for 3: $1+1 = 2 < \infty$

0	0	T	null
1	10	F	0
2	1	T	0
3	2	F	2

not all known

- unknown with min dist = 3

0	0	T	null
1	10	F	0
2	1	T	0
3	2	T	2

  - 3 is connected to 2,1 only 1 is unknown
  - for 1: 2+1 = 3 < 10

0	0	T	null
1	3	F	0
2	1	T	0
3	2	T	2

not all known

- unknown with min dist = 1

0	0	T	null
1	10	F	0
2	1	T	0
3	2	T	2

    - 1 is not connected to any unknown things

- done

All-pair shortest path
→ run Dijkstra |V| times → O(|V|^3)

Acyclic graphs: critical path analysis

- dijkstra can be done in one pass
- running time: O(|E|+|V|)
- activity-node graph
  - node → an activity that must be performed, along with time it takes to complete the activity
  - edge → precedence relationship: (v,w) = activity v must be completed before activity w may begin
- assume any activities that do not depend (either directly or indirectly) on each other can be performed in parallel by different servers.
- Ans.
  - what is the earliest completion time for the project
  - which activities can be delayed, and by how long, without affecting the minimum completion time
- **event-node graph** →
  - **event** → the completion of an activity and all its dependent activities
  - events reachable from a node v may not commence until after the event v is completed
  - dummy edges (weight 0) and nodes may need to be inserted in the case where an activity depends on several others
- find the length of the longest path from the first event to the last event

- v.E = earliest completion time @ node v

```
startVertex.E = 0;
startVertex.known = true;
while(not all known)
{
    for all unknown vertices v
    {
        if(all parents w of v are known) //w→v
        {
            v.E = $\max_{(w,v)\in Edge}\left(w.E + c_{w,v}\right)$
            v.know = true
        }
    }
}
```

- v.L = the latest time that each event can finish without affecting the final completion time

```
    finalVertex.L = finalVertex.E;
    finalVertex.known = true;
    while(not all known)
    {
        for all unknown vertices v
        {
            if(all children w of v are known) //v→w
            {
                v.E = $\min_{(v,w)\in Edge}(w.L - c_{v,w})$
                v.know = true
            }
        }
    }
```

- slack time for each edge
  = the amount of time that the completion of the corresponding activity can be delayed without delaying the overall completion
  $slack_{v,w} = L_w - E_v - c_{v,w}$; v→w
  because
  v have to be reached before day E_v
  w don't have to be reached until day L_w
  so, the time to do the edge = $L_w - E_v$

- activity that has 0 slack → critical activity, must finish on schedule
- **critical path** → (at least one for graph) consist entirely of zero-slack edges

Maximum-Flow Algorithm

- edge capacities $c_{v,w}$
- Vertex s → source; t → sink
- through any edge (v,w), at most $c_{v,w}$ units of flow may pass
- at any vertex v that is not s or t, the total flow coming in must equal the total flow going out
- maximum flow problem → determine the maximum amount of flow that can pass from s to t
- directed
- acyclic is not a requirement

- have 3 pictures at each stage
  - G : the original graph, telling capacity, never changed
  - G_f : the flow used currently
  - G_r : **residual graph** → how much more flow can be added
    - **residual edge**
    - each edge → 2 residual edges
      - same direction: amount that can send more = capacity - currentAmount
      - opposite direction: amount that can send back = current amount (but opposite direction)
        → allow the algorithm to undo its decisions

    don't draw 0 edges
- at each stage, find a path from s to t in G_r → **augmenting path**
  - nondeterministic
  - minimum edge on this path is the amount of flow that can be added to every edge on the path
  - determine by
    - unweighted shortest-path algorithm
    - or modified Dijkstra to find one that allows the largest increase in flow
  - adjust G_f
  - recomputed G_r
- when there is no path from s to t → done

- if the edge capacities are rational numbers, this algorithm always terminates with a maximum flow

Minimum Spanning Tree

of an undirected graph
⇒ a tree formed from graph edges that connected all the vertices of G at lowest total cost

Spanning tree

- exists if and only if G is connected
- minimum $|E| = |V| - 1$
- acyclic → tree
- spanning → covers every vertex
- if an edge e, that is not in the ST is added
  → a cycle is created
  - removal of any edge on the cycle
    - reinsates the spanning tree property
    - if e has lower cost than the edge the was removed → cost of spanning tree is lowered
- Add edge when a spanning tree is created
  - if edge is one of the minimum cost that avoids creation of a cycle
    → cost of the resulting spanning tree cannot be improved
    - because any replacement edge would have cost at least as much as an edge already in the spanning tree

Prim's Algorithm

- run on undirected graph
- have to put every edge in 2 adjacency list
- d_v = the weight of the shortest edge connecting v to a known vertex
- p_v = the last vertex to cause a change in d_v

v	known	d_v	p_v

```
choose any vertex to be starting point
    o   dv = 0
    o   pv = itself
//the tree (of known vertex) grows by 1 every loop
while(not all known)
{
    find unknown vertex v with min dist
    mark it known
    for each unknown vertex w adjacent to v
    {
        if(cw,v < current dw)
        {
            dw = cw,v;
            set w.via = v;
        }
    }
}
```

- At any stage in the algorithm, we find a new vertex to add to the tree by choosing the edge (u,v) such that the cost of (u,v) is the smallest among all edges where u is in the tree and v is not
- each step adds one edge and one vertex to the tree
- running time
  $= O(|V|^2)$ without heap → optimal for dense graphs
  $= O(|E|\log|V|)$ using binary heaps

Kruskal's Algorithm

- maintains a forest → collection of trees

```
public void kruskal()
{
    int edgesAccepted = 0;
    DisjSet s;
    PriorityQueue h; //using heap
    Vertex u,v;
    SetType uSet, vSet;
    Edge e;

    //create heap from weight of edges
    h = readGraphIntoHeapArray();
    h.buildHeap();

    //initially there are |V| single-node trees
    //each vertex is initially in its own set
    s = new DisjSet( NUM_VERTICES );

    //terminate when enough edges are accepted
    //required |V|-1 edges
    while( edgesAccepted < NUM_VERTICES - 1 )
    {
        //select the edge in order of smallest weight
        e = h.deleteMin();
        //make edge e = (u,v)
        uSet = s.find(u);
        vSet = s.find(v);
        //accept an edge if it does not cause a cycle
        if(uSet != vSet)
        {
            //accept the edge
            edgesAccepted++;
            //adding an edge merges 2 trees into one
            s.union(uSet,vSet);
        }
    }
    //when the algorithm terminates,
    //there is only one tree → minimum spanning tree
}
```

- u and v belong to the same set
  if and only if
  they are connected in the current spanning forest
- if u and v are in the same set, → the edge is rejected ← since they are already connected, adding (u,v) would form a cycle

- worst-case running time → $O(|E|\log|E|)$ ← dominated by the heap operations
  $= O(|E|\log|V|)$ since $|E| = O(|V|^2)$

Application
- wire a house with a minimum of cable

Summary: Application of graph

edge: distance, time, cost, capacity
node: activity, location

- Dijkstra

```
while(not all known)
{
    find unknown with min dist v
    mark it known
    find all adjacent vertices
    for each unknown adjacent vertices w
    {
        if(v.dist + weight of edge (v,w) < w.dist )
        {
            update dis w
            set w.via = v
        }
    }
}
```

- critical path analysis
  - edge: task , weight: time this task take
  - node: completion of edge before it

```
startVertex.E = 0;
startVertex.known = true;
while(not all known)
{
    for all unknown vertices v
    {
        if(all parents w of v are known) //w→v
        {
            v.E = $\max_{(w,v)\in Edge}\left(w.E + c_{w,v}\right)$
            v.know = true
        }
    }
}
```

Sorting

Insertion sort

- $O(N^2)$

Bucket sort

- have N integers in the range 1 to M (or 0 to M-1)
- keep an array called count[M], initialized to 0
- When A_i is read, count[A_i]++.
- After all the input is read, scan the count array, printing out a

representation of the sorted list

- O(M+N)

Radix sort

- use several passes of bucket
- bucket-sort by the least significant "digit" significant
- more than one number could fall into the same bucket and these numbers could be different → keep them in a **list**
- Bad idea to use array since all number can have a digit in common (worse case for space), this will make a bucket of that digit becomes size = N. Don't know which one → every bucket's size = N → space = O(N^2)
- when several numbers enter a bucket, they enter in sorted order
- running time → O(P(N+B)) → O(N)
    - P = number of passes (digit), usually constant
    - N = number of element to sort
    - B = number of bucket, usually << N

Mergesort

- worst-case O(N log N)

merge: O(N) ← at most N-1 comparisons are made

- takes 2 input arrays A and B, an output array C, and three counters, Actr, Bctr, and Cctr, which are initially set to the beginning of their respective arrays
- the smaller of A[Actr] and B{Bctr] is copied to the next entry in C, and the appropriate counter are advanced.
- When either input list is exhausted, the remainder of the order list is copied to C

```
static void merge(Comparable[] a, Comparable[] tmpArray,
    int leftPos, int rightPos, int rightEnd)
{
    //leftPos, rightPos → where we're considering now
    int leftEnd = rightPos – 1;
    int tmpPos = leftPos;
        //will denote position in the tmpArray considered now
    int numElements = rightEnd – leftPos + 1;

    while(leftPos<=leftEnd && rightPos<=rightEnd)
    {
        if(a[leftPos].compareTo(a[rightPos])<=0)
            tmpArray[tmpPos++] = a[leftPos++];
        else
            tmpArray[tmpPos++] = a[rightPos++];
    }

    //one of the following 2 while loop will actually used
    //it is used to
    //copy the rest of the "either" left "or" right half
    while(leftPos<=leftEnd)
    {
        tmpArray[tmpPos++] = a[leftPos++];
    }
    while(rightPos<=rightEnd)
    {
        tmpArray[tmpPos++] = a[rightPos++];
    }

    //copy tmpArray back
    for(int i=0; i<numOfELements; i++, rightEnd--)
    {
        a[rightEnd] = tmpArray[rightEnd];
    }
}
```

```
static void mergeSort(Comparable[] a)
{
    Comparable[] tmpArray = new Comparable[a.length];
    mergeSort(a, tmpArray, 0, a.length-1);
}
```

```
static void mergeSort(Comparable[] a, Comparable[] tmpArray,
        int left, int right)
{
    if(left<right)
    {
        int center = (left+right)/2;
        mergeSort(a, tmpArray, left, center);
        mergeSort(a, tmpArray, center+1, right);
        merge(a, tmpArray, left, center+1, right);
    }
}
```

- T(1) = 1, T(N) = 2T(N/2) + N → T(N) = N log N + N = O(N log N)

- use O(N) extra memory

Quicksort

- fastest known sorting algorithm in practice
- average: O(N log N)
- worst case: O(N^2)

- If the number of elements in S is 0 or 1, then return
- Pick any element v in S → **pivot**
    - works no matter which element is chosen
    - good choice → median of first,center,last

- Partition S-{v} into 2 disjoint groups:
    - $S_1=\{x\in S-\{v\}|x\leq v\}$
    - $S_2=\{x\in S-\{v\}|x\geq v\}$
- Return {quicksort(S_1) followed by v followed by quicksort(S_2)}