Princeton Algorithm Coursera Course Notes

2021 – 04 – 06

Junfan Zhu

(junfanz@gatech.edu; junfanzhu@uchicago.edu)

Course Links

Algorithm Part I and Part II, by Princeton University Robert Sedgewick and Kevin Wayne.

> Char 1-6: https://www.coursera.org/learn/algorithms-part1/
>
> Char 7-12: https://www.coursera.org/learn/algorithms-part2/

Table of Contents

---

1. Union Find

Find connectivity

```
public class UF
    void union
    boolean connected
```

1.1. Quick Find (eager approach)

$O(N^2)$, too slow

- `Find`
  - Check if p and q have same id.
- `Union`
  - Merge components containing p and q, change entries whose id equals $id[p]$ to $id[q]$.

1.2. Quick Union (lazy approach)

- $id[i]$ is parent of i
- Root of i is $id[id[...id[i]..]]$.
- `Find`
  - Check if p and q have same id.
- `Union`
  - Merge components containing p and q, set the id of p's root to id of q's root.

Advantage: Only one value changes in the array.

Improvement:

- Weighted Quick Union: $N + M \log N$
  - In Union step, need to change the size of array, so that depth of any node x is at most $\lg N$.
- Weighted Quick Union + Path Compression: $N + M \lg N$

1.3. Union Find Applications

Percolation (Phase Transition)

---

2. Stack and Queues

2.1. Stack

2.1.1. Linked List Implementation

```
public class LinkedStackOfStrings
{
    private Node first = null;
    private class Node
    {
        String item;
        Node next;
    }
    public boolean isEmpty()
    { return first == null;}

    public void push(String item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }
    public String pop()
    {
        String item = first.item;
        first = first.next;
        return item;
    }
}
```

2.1.2. Array implementation

```
public class FixedCapacityStackOfStrings
{
    private String[] s;
```

```java
    private int N=0;

    public FixedCapacityStackOfStrings(int capacity) // a cheat (stay tuned)
    { s = new String[capacity]; }

    public boolean isEmpty()
    {return N == 0;}

    public void push(String item)
    { s[N++] = item; } // s[N++] use to index into array; then increment N

/*
    public String pop()
    {return s[--N];} // Decrement N; then use index into array
    // Loitering: Holding a reference to
    // an object when it's no longer needed.

*/

    public String pop()
    {
        String item = s[--N];
        s[N] = null;
        return item;
    // This time avoids loitering. Garbage collector
    // can reclaim memory only if no outstanding references.
    }

}
```

2.1.3. Resizing-array vs. Linked-list

- Linked-list
  - Every operation takes const time in the **worst case**.
  - Uses extra time and space to deal with the links.
- Resizing-array
  - Every operation takes const **amortized** time.
  - Less wasted space.

2.2. Queue (Linked List implementation)

```java
public class LinkedQueueOfStrings
{
```

```
    private Node first, last;
    private class Node
    { /* same as StackOOfStrings*/ }

    public boolean isEmpty()
    { return first == null;}

    public void enqueue(String item)
    {
        Node oldlast = last;
        last = new Node();
        last.item = item;
        last.next = null;
        if (isEmpty()) first = last;
        else           oldlast.next = last;
    }

    public String dequeue()
    {
        String item = first.item;
        first = first.next;
        if (isEmpty()) last = null;
        return item;
    }
}
```

2.3. Generics

2.3.1. Generics Intro

Apart from `StackOfStrings`, we also want `StackOfURLs`, `StackOfInts`, ...

Attempt 1 (not good): Implement a stack with items of type `Object`.

- Casting is required in client.
- Casting is error-prone: run-time error if types mismatch.

```
StackOfObjects s = new StackOfObjects();
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b);
a=(Apple) (s.pop()); // run-time error
```

Attempt 2 (Java Generics)

- Avoid casting in client.
- Discover type mismatch errors at compile-time instead of run-time.

```
Stack<Apple> s = new Stack<Apple>(); // Apple is type parameter
Apple a = new Apple();
Orange b = new Orange();
s.push(a);
s.push(b); // compile time error. But we welcome
// compile time errors, avoid run-time errors.
a = s.pop();
```

2.3.2. Generic stack: linked-list implementation

```
public class Stack<Item> // replace StringName with Item,
// which is generic type name
{
    private Node first = null;
    private class Node
    {
        Item item;
        Node next;
    }
    public boolean isEmpty()
    {  return first == null; }
    public void push(Item item)
    {
        Node oldfirst = first;
        first = new Node();
        first.item = item;
        first.next = oldfirst;
    }
    public Item pop()
    {
        Item item = first.item;
        first = first.next;
        return item;
    }
}
```

2.3.3. Generic Stack: array implementation

```
public class FixedCapacityStack<item>
{
    private Item[] s;
```

```java
    private int N = 0;
    public FixedCapacityStack(int capacity)
    // {s = new Item[capacity];}
    // generic array creation is not allowed in Java
    {s = (Item[]) new Object[capacity]; } // It's the ugly unchecked cast
    // but works, and will output 1 warning.

    public boolean isEmpty()
    { return N == 0; }

    public void push(Item item)
    { s[N++] = item; }

    public Item pop()
    { return s[--N];}
}
```

2.3.4. Generic data types: autoboxing

What to do about primitive types?

- Wrapper type.
  - Each primitive type has a **wrapper** object type.
  - Ex: `Integer` is wrapper type for `int`.
- Autoboxing.
  - Automatic cast between a primitive type and its wrapper.
- Syntatic suger.
  - Behind-the-scenes casting.

```java
Stack<Interger> s = new Stack<Integer>();
s.push(19); // s.push(new Integer(19));
int a = s.pop(); // int a = s.pop().intValue();
```

- Bottom line.
  - Client code can use generic stack for **any** type of data.

2.4. Iteration

Design challenge. Support iteration over stack items by client, without revealing the internal representation of the stack.

Java solution. Make stack implement the **Iterable** interface.

Iterable

A method that returns an **Iterator**.

```
// Iterable interface
public interface Iterable<Item>
{
    Iterator<Item> iterator();
}
```

2.4.1. Iterator

Has methods **hasNext()** and **next()**.

```
// Iterator interface
public interface Iterator<Item>
{
    boolean hasNext();
    Item next();
    // void remove(); // optional
}

//Example: For each
for (string s:stack)
    StdOut.println(s);

// equivalent:
Iterator<String> i = stack.iterator();
while (i.hasNext())
{
    String s = i.next()
    StdOut.println(s);
}
```

2.4.2. Stack Iterator: linked-list implementation

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item>
{
    // omitted a lot...
    public Iterator<Item> iterator() {
        return new ListIterator();
    }
```

```
    private class ListIterator implements Iterator<Item>
    {
        private Node current = first;
        public boolean = hasNext() {
            return current != null;
        }
        public void remove() { /* throw unsupported OperationException */}
        public Item next() // throw NoSuchElementException
        // if no more items in iteration
        {
            Item item = current.item;
            current = current.next;
            return item;
        }
    }
}
```

2.4.3. Stack Iterator: Array implementation

```
import java.util.Iterator;
public class Stack<Item> implements Iterable<Item>
{
    // omitted a lot...
    public Iterator<Item> iterator()
    { return new ReverseArrayIterator();}
    private class ReverseArrayIterator implements Iterator<Item>
    {
        private int i = N;
        public boolean hasNext() {return i > 0;}
        public void remove() {/* not supported*/}
        public Item next() { return s[--i];}
    }
}
```

2.5. Miscellaneous to 2.4

2.5.1. Function Calls

How a compiler implements a function?

- Function call: **push** local environment and return address.
- Return: **pop** return address and local environment.

Recursive function.

- Function that calls itself (can always use an explicit stack to remove recursion.)

2.5.2. Dijkstra's Two-stack Algorithm

2.6. Sort

2.6.1. Callbacks

Goal. Sort **any** type of data.

Q: How can `sort()` know how to compare data of type `Double`,`String`, and `java.io.File` without any info about the type of an item's key?

Callback = reference to executable code.

- Client passes array of objects to `sort()` function.
- The `sort()` function calls back object's `compareTo()` method as needed.

Implementing Callbacks

- Java: interfaces
- C: function pointers
- C++: class-type functors
- Python: first-class functions.

Callback Roadmap.

```
// Part-1: client
import java.io.File;
public class FileSorter
{
    public static void main(String[] args)
    {
        File directory = new File(args[0]);
        File[] files = directory.listFiles();
        Insertion.sort(files);
        for (int i = 0; i < files.length; i++)
            StdOOut.println(files[i].getName());
    }
}
// Part-2: Comparable interface (built in to Java)
public interface Comparable<Item>
{
```

```java
    public int compareTo(Item that);
}

// Part-3: object info
public class File
implements Comparable<File>
{
    // omitted a lot..
    public int compareTo(File b)
    {
        return -1;
        return +1;
        return 0;
        // ... //
    }
}

// Part-4: Sort implementation
public static void sort(Comparable[] a)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0; j--)
            if (a[j].compareTo(a[j-1]) < 0)
                exch(a,j,j-1);
            else break;
}
```

2.6.2. Implement Comparable interface

Date data type. Simplified version of `java.util.Date`.

```java
public class Date implements Comparable<Date>
{ // only compare dates to other dates
    private final int month, day, year;
    public Date(int m, int d, int y)
    {
        month = m;
        day = d;
        year = y;
    }
    public int compareTo(Date that)
    {
        if (this.year < that.year) return -1;
        if (this.year > that.year) return +1;
```

```
        if (this.month < that.month) return -1;
        if (this.month > that.month) return +1;
        if (this.day < that.day) return -1;
        if (this.day > that.day) return +1;
        return 0;
    }
}
```

2.6.3. Two sorting

`Less`. v $<$ w?

```
private static boolean less(Comparable v, Comparable w)
{
    return v.compareTo(w) < 0;
}
```

`Exchange`. Swap item in array a[] at index i with the one at index j.

```
private static void exch(Comparable[] a, int i, int j)
{
    Comparable swap = a[i];
    a[i] = a[j];
    a[j] = swap;
}
```

2.7. Selection Sort

$O(N^2)$

Algorithm. ↑ sort from left to right.

Invariants.

- Entries the left of ↑ fixed and in ascending order.
- No entry to the right of ↑ is smaller than any entry to the left of ↑.

2.7.1. Selection Sort Inner Loop

- Move the pointer to the right.

```
i++;
```

- Identify index of miniimum entry on the right.

```
int min = i;
for (int j = i+1; j < N; j++)
    if (less(a[j], a[min]))
        min = j;
```

- Exchange into position.

```
exch(a,i,min);
```

2.8. Insertion Sort

$O(N^2)$

In insertion i, swap $a[i]$ with each larger entry to its left.

Implementation

```
public class Insertion
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        for (int i = 0; i< N; i++)
            for (int j = i; j > 0; j--)
                if (less(a[j],a[j-1]))
                    exch(a,j,j-1);
                else break;
    }
    private static boolean less(Comparable v, Comparable w)
    {/*...*/}
    private static void exch(Comparable[] a, int i, int j)
    {/*...*/}
}
```

2.9. Shell Sort

Fast. Worst case of 3x+1: $O(N^{3/2})$. Accurate model has not yet been discovered.

Move entries more than one position at a time by h-sorting the array.

Prop. A g-sorted array remains g-sorted after h-sorted it.

Why insertion sort?

- Big increments ⇒ small subarray.
- Small increments ⇒ nearly in order [stay tuned].

Implementation

```java
public class Shell
{
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        int h = 1;
        while (h<N/3) h - 3*h +1; // 1,4,13,40,121,364,...
        while (h>=1)
        { //h-sort the array
            for (int i = h; i < N; i++) // inversion sort
            {
                for (int j = i; j >=h && less(a[j], a[j-h]); j-=h)
                    exch(a,j-h);
            }
            h = h/3; // move to next increment
        }
    }
    private static boolean less(Comparable v, Comparable w)
    { /*...*/}
    private static void exch(Comparable[] a, int i, int j)
    { /*...*/}
}
```

2.10. Shuffle Sort

Generate a random number for each array entry. Sort the array.

Goal. Rearrange array so that result is a uniformly random permutation **in linear time**.

---

3. Merge Sort and Quick Sort

3.1. Merge Sort

Upper bound and lower bound are both $O(N \log N)$, which proves Merge Sort is the optimal algorithm.

Plan

- Divide array into 2 halves
- **Recursively** sort each half
- Merge 2 halves

Merging

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    // Assert statement throws exception unless boolean condition is true.
    // Can enable or disable assertions at any time, no cost in production code
    // Use assertion to check internal invariants,
    // assume assertions will be disabled in production code.
    assert isSorted(a,lo,mid); // precondition: a[lo..mid] sorted
    assert isSorted(a,mid+1,hi); // precondition: a[mid+1..hi] sorted

    for (int k = lo; k <= hi; k++) // copy
        aux[k] = a[k];

    int i = lo, j = mid+1; // merge
    for (int k = lo; k <=hi; k++)
    {
        if (i > mid)        a[k] = aux[j++];
        else if (j > hi)    a[k] = aux[i++];
        else if (less(aux[j], aux[i])) a[k] = aux[j++];
        else                a[k] = aux[i++];
    }
    assert isSorted(a,lo,hi); // postcondition: a[lo..hi] sorted
}
```

Merge Sort

```
public class Merge
{
    private static void merge(...)
    { /*as before*/}
    private static voide sort(Comparable[] a, Comparable[] aux, int lo, int hi)
    {
        if (hi <= lo) return;
        int mid = lo + (hi-lo)/2;
        sort (a, aux, lo, mid);
        sort (a, aux, mid+1, hi);
        sort (a, aux, mid+1, hi);
        merge(a, aux, lo, mid, hi);
    }
    public static void sort(Comparable[] a)
```

```
    {
        aux = new Comparable[a.length];
        sort(a, aux, 0, a.length - 1);
    }
}
```

Merge Sort Improvements

Elimiate the copy to the auxiliary array. Save time (but not space) by switching the role of the input and auxiliary array in each recursive call.

```
private static void merge(Comparable[] a, Comparable[] aux, int lo, int mid, int hi)
{
    int i = lo, j = mid+1;
    for (int k = lo; k <= hi; k++)
    {// merge from a[] to aux[]
        if (i>mid)      aux[k] = a[j++];
        else if (j>hi)  aux[k] = a[i++];
        else if (less(a[j], a[i])) aux[k] = a[j++];
        else            aux[k] = a[i++];
    }
}
private static void sort(Comparable[] a, Comparable[] aux, int lo, int hi)
{ // sort(a) initializes aux[] and sets aux[i] = a[i] for each i
    if (hi <= lo) return;
    int mid = lo + (hi-lo)/2;
    sort(aux, a, lo, mid);
    sort(aux, a, mid+1, hi);
    merge(a, aux, lo, mid, hi);
}
```

3.2. Bottom-up Merge Sort

```
public class MergeBU
{
    private static Comparable[] aux;
    private static void merge(Comparable[] a, int lo, int mid, int hi)
    {/* as before */}
    public static void sort(Comparable[] a)
    {
        int N = a.length;
        aux = new Comparable[N];
        for (int sz = 1; sz < N; sz = sz+sz)
            for (int lo = 0; lo < N-sz; lo+=sz+sz)
                merge(a, lo, lo+s-1, Math.min(lo+sz+sz-1, N-1));
```

```
    }
}
```

3.3. Comparators

Comparator interface using sorting libraries.

- Use `Object` instead of `Comparable`.
- Pass `Comparator` to `sort()` and `less()` and use it in `less()`.

Insertion sort with Comparator

```
public static void sort(Object[] a, Comparator comparator)
{
    int N = a.length;
    for (int i = 0; i < N; i++)
        for (int j = i; j > 0 && less(comparator, a[j], a[j-1]); j--)
            exch(a,j,j-1);
}
private static boolean less(Comparator c, Object v, Object w)
{ return c.compare(v,w) < 0;}
private static void exch(Object[] a, int i, int j)
{ Object swap = a[i]; a[i] = a[j]; a[j] = swap;}
```

Comparator interface

- Define a nested class that implements the `Comparator` interface.
- Implement the `compare()` method.

```
public class Student
{
    public static final Comparator<Student> BY_NAME = new ByName();
    /* one Comparator for the class*/
    public static final Comparator<Student> BY_SECTION = new BySection();
    private final String name;
    private final int section;

    private static class ByName implements Comparator<Student>
    /* one Comparator for the class*/
    {
        public int compare(Student v, Student w)
        { return v.name.compareTo(w.name);}
    }
```

```java
    private static class BySection implements Comparator<Student>
    {
        public int compare(Student v, Student w)
        { return v.section - w.section;}
        // no danger of overflow
    }

}
```

3.4. Stability

Which sorts are stable?

- Insertion sort and merge sort: ✓ stable
- Selection sort and shell sort: × not stable

Explanation:

- Insertion sort: equal items never move past each other.
- Merge sort: takes from left subarray if equal keys.
- Selection sort, Shell sort: Long distance exchange may move an item past some equal item.

3.5. Quick Sort

$O(N \log N)$ on average, # of compares is 39% more than mergesort, but faster than mergesort because of less data movement. May go **quadratic** if array is sorted or reverse sorted, or has many duplicates (even if randomized).

Quick sort is not stable. It's in-place sorting algorithm. (More improvements see lecture, like median or cutoff.)

Step

- Shuffle the array.
- Partition, so that for some `j`:
  - entry `a[j]` is in place
  - no larger entry to the left of `j`
  - no smaller entry to the right of `j`
- Sort each piece recursively

  Quick sort for partitioning

```java
private static int partition(Comparable[] a, int lo, int hi)
{
    int i = lo, j = hi+1;
    while (true)
    {
        while (less(a[++i], a[lo]))
            if (i == hi) break; // find item on left to swap
        while (less(a[lo], a[--j]))
            if (j == lo) break; // find item on right to swap
        if (i >= j) break; // check if pointers cross
        exch(a,i,j); // swap
    }
    exch(a, lo, j); //swap with partitioning item
    return j; // return index of item now known to be in place
}
```

Quick sort

```java
public class Quick
{
    private static int partition(Comparable[] a, int lo, int hi)
    {/* see above*/}
    public static void sort(Comparable[] a)
    {
        StdRandom.shuffle(a);
        sort(a,0,a.length-1);
        // shuffle needed for performance guarantee (stay tuned)
    }
    private static void sort(Comparable[] a, int lo, int hi)
    {
        if (hi <= lo) return;
        int j = partition(a, lo, hi);
        sort(a, lo, j-1);
        sort(a, j+1, hi);
    }
}
```

3.6. Selection (Find k-th largest)

Quick-select

On average $O(N)$

Partitioon array so that:

- Entry `a[j]` is in place.

- No larger entry to the left of j
- No smaller entry to the right of j

Repeat in **one** subarray, depending on j; finished when j equals k.

```
public static Comparable select(Comparable[] a, int k)
{
    StdRandom.shuffle(a);
    int lo = 0, hi = a.length - 1;
    while (hi > lo)
    {
        int j = partition(a,lo,hi);
        if (j<k)        lo = j + 1;
        else if (j>k)   hi = j - 1;
        else            return a[k];
    }
    return a[k];
}
```

3.7. Duplicate Keys

Merge sort with duplicate keys. $O(N \log N)$

Quicksort with duplicate keys. qsort. **Quadratic** unless partitioning stops on equal keys.

---

4. Priority Queues

Find the largest M items in a stream of N items.

```
MinPQ<Transaction> pq = new MinPQ<Transaction>();
// use a min-oriented pq
// Transaction data type is Comparable ordered by $$
while (StdIn.hashNextLine())
{
    String line = StdIn.readLine();
    Transaction item = new Transaction(line);
    pq.insert(item);
    if (pq.size() > N) // pq contains largest M items
        pq.swlMin();
}
```

- sort: time $O(N \log N)$, space $O(N)$
- elementary PQ: time $O(MN)$, space $O(M)$
- binary heap: time $O(N \log M)$, space $O(M)$
- best in theory: time $O(N)$, space $O(M)$

Priority queue: unordered array implementation

```
public class UnorderedMaxPQ<Key extends Comparable<Key>>
{
    prvate Key[] pq; // pq[i] = ith element on pq
    private int N; // number of elements on pq

    public UnorderedMaxPQ(int capacity)
    { // no generic array creation
        pq = (Key[]) new Comparable[capacity];
    }
    public boolean isEmpty()
    {  return N == 0; }
    public void insert(Key x)
    {  pq[N++] = x;}
    public key delMax()
    {
        int max = 0;
        for (int i = 1; i < N; i++)
            if (less(max,i)) max = i;
        exch(max, N-1);
        // less() and exch(), similar to sorting methods
        return pq[--N];
    }
}
```

4.1. Binary Heaps

Promotion in the Heap. When child's key is larger than parents' key:

- Exchange key in child with key in parent.
- Repeat until heap order restored.

```
private void swim(int k)
{
    while (k > 1 && less(k/2, k))
    { // parent of node at k is at k/2
        exch(k,k/2);
        k = k/2;
```

```
    }
}
```

Insertion in the Heap. $O(\lg N)$. Add node at end and swim it up.

Demotion in the Heap. Parents' key becomes **smaller** than one or both of its children's.

- Exchange key in parent with key in larger child.
- Repeat until heap order restored.

```
private void sink(int k)
{
    while (2*k <=N)
    {
        int j = 2*k;
        if (j < N && less(j,j+1)) j++;
        // children of noded at k are 2k and 2k+1
        if (!less(k,j)) break;
        exch(k,j);
        k = j;
    }
}
```

Delete Max in the Heap. Exchange root with node at end, then sink it down. $O(2 \lg N)$

Binary Heap

```
public class MaxPQ<Key extends Comparable<Key>>
{
    private Key[] pq;
    private int N;

    public MaxPQ(int capacity)
    { pq = (Key[]) new Comparable[capacity+1];}

    //PQ ops
    public boolean isEmpty()
    {  return N == 0; }
    public void insert(Key key)
    public Key delMax()
    {/*see above*/}

    // heap helper functions
```

```
    private void swim(int k)
    private void sink(int k)
    {/*see above*/}

    //array helper functions
    private boolean less(int i, int j)
    { return pq[i].compareTo(pq[j]) < 0;}
    private void exch(int i, int j)
    { Key t = pq[i]; pq[i] = pq[j]; pq[j] = t;}
}
```

4.1.1. Immutability

Can't change the data type once created. Safe to use as key in priority queue or symbol table. Classes should be immutable.

4.2. Heap Sort

- Heap construction: $O(N)$ compares and exchanges
- Heap sort: $O(N \lg N)$ compares and exchanges

Idea

- Create max-heap with all N keys.
- Repeatedly remove the maximum key.

Steps

1. Build max heap using bottom-up method.
2. Remove max, one at a time. Leave in array instead of nulling out.

```
public class Heap
{
    public static void sort(Comparable[] pq)
    {
        int N = pq.length;
        for (int k = N/2; k >= 1; k--)
            sink(pq,k,N);
        while (N>1)
        {
            exch(pq,1,N);
            sink(pq,1,--N);
        }
```

```
    }
    // omitted...
}
```

Complexity

In-place sorting algorithm with $O(N \log N)$ worst-case.

- Mergesort: no, linear extra space.
- Quicksort: no, quadratic time in worst case.
- Heapsort: yes!

Heapsort is optimal for both time and space, *but*:

- Inner loop longer than quicksort's.
- Make poor use of cache memory.
- Not stable.

4.2.1. Sort Algorithms Summary

Sorting algorithms: summary

	inplace?	stable?	worst	average	best	remarks
selection	x		$N^2 / 2$	$N^2 / 2$	$N^2 / 2$	N exchanges
insertion	x	x	$N^2 / 2$	$N^2 / 4$	N	use for small N or partially ordered
shell	x		?	?	N	tight code, subquadratic
quick	x		$N^2 / 2$	2 N ln N	N lg N	N log N probabilistic guarantee fastest in practice
3-way quick	x		$N^2 / 2$	2 N ln N	N	improves quicksort in presence of duplicate keys
merge		x	N lg N	N lg N	N lg N	N log N guarantee, stable
heap	x		2 N lg N	2 N lg N	N lg N	N log N guarantee, in-place
???	x	x	N lg N	N lg N	N lg N	holy sorting grail

Figure 1: Sort Summary

4.3. Symbol Tables

Associative array abstraction. Associate one value with each key.

- Values are not `null`.
- Method `get()` returns `null` if key not present.
- Method `put()` overwrites old value with new value.

4.3.1. Binary Search

```java
public Value get(Key key)
{
    if (isEmpty()) return null;
    int i = rank(key);
    if (i < N && keys[i].compareTo(key) == 0) return vals[i];
    else return null;
}
private int rank(Key key)
// number of keys < key
{
    int lo = 0, hi = N-1;
    while (lo <= hi)
    {
        int mid = lo + (hi - lo)/2;
        int cmp = key.compareTo(keys[mid]);
        if (cmp < 0)       hi = mid - 1;
        else if (cmp > 0)  lo = mid + 1;
        else               return mid;
    }
    return lo;
}
```

4.4. Binary Search Trees (BST)

Def. BST is a **binary tree** in **symmetric order**.

A binary tree is either:

- Empty.
- Two disjoint binary trees (left and right).

Symmetric order. Each node has a key, and every node's key is:

- Larger than all keys in its left subtree.

- Smaller than all keys in its right subtree.

A BST is a reference to a root `Node`.

- `Key`
- `Value`
- A reference to the left (smaller keys) and right (larger keys) subtree.

```
private class Node
{
    private Key key;
    private Value val;
    private Node left, right;
    public Node(Key key, Value val)
    {
        this.key = key;
        this.val = val;
    }
}
```

4.4.1. BST Search

$O(\lg N)$

```
public Value get(Key key)
{
    Node x = root;
    while (x != null)
    { // return value corresponding to given key
        int cmp = key.compareTo(x.key);
        if      (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else              return x.val;
    }
    return null; // if no such key return null
}
```

4.4.2. BST Insert

$O(\lg N)$

Concise but tricky recursive Put: associate value with key.

```java
public void put(Key key, Value val)
{ root = put(root, key, val); }
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key,val);
    int cmp = key.compareTo(x.key);
    if (cmp < 0)
        x.left = put(x.left, key, val);
    else if (cmp > 0)
        x.right = put(x.right, key, val);
    else
        x.val = val;
    return x;
}
```

Tree shape depends on order of insertion.

Correspondence between BST and quicksort partitioning is 1-1 if array has no duplicate keys.

4.4.3. Ordered Operations in BSTs

$O(h)$ (height of the tree, in proportion to $\log N$)

Compute the `floor` in the BST.

```java
public Key floor(Key key)
{
    Node x = floor(root, key);
    if (x == null) return null;
    return x.key;
}
private Node floor(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);

    if (cmp == 0) return x;
    if (cmp < 0) return floor(x.left, key);

    Node t = floor(x.right, key);
    if (t != null) return t;
    else           return x;
}
```

BST Subtree Counts

```java
private class Node
{
    private Key key;
    private Value val;
    private Node left;
    private Node right;
    private int count; // number of nodes in subtree
}
public int size()
{ return size(root); }
private int size(Node x)
{
    if (x == null) return 0;
    return x.count;
}
private Node put(Node x, Key key, Value val)
{
    if (x == null) return new Node(key, val, 1);
    int cmp = key.compareTo(x.key);
    if (cmp < 0) x.left = put(x.left, key, val);
    else if (cmp > 0) x.right = put(x.right, key, val);
    else x.val = val;
    x.count = 1 + size(x.left) + size(x.right);
    return x;
}
```

Rank in BST (Recursive algorithm with 3 cases)

```java
public int rank(Key key)
{ return rank(key, root); }
private int rank(Key key, Node x)
{
    if (x == null) return 0;
    int cmp = key.compareTo(x.key);
    if  (cmp < 0) return rank(key, x.left);
    else if (cmp > 0) return 1 + size(x.left) + rank(key, x.right);
    else return size(x.left);
}
```

4.4.4. Deletion in BST

Hibbard deletion is unastisfactory, because not symmetric, and not random ($O(\sqrt{N})$). Open problem. And that's why we need red-black BST.

```java
public void delete(Key key)
{ root = delete(root, key); }
private Node delete(Node x, Key key)
{
    if (x == null) return null;
    int cmp = key.compareTo(x.key);
    // search for key
    if (cmp < 0) x.left = delete(x.left, key);
    else if (cmp > 0) x.right = delete(x.right, key);
    else{
        // no right child
        if (x.right == null) return x.left;
        // no left child
        if (x.left == null) return x.right;
        // replace with successor
        Node t = x;
        x = min(t.right);
        x.right = deleteMin(t.right);
        x.left = t.left
    }
    // update subtree counts
    x.count = size(x.left) + size(x.right) + 1;
    return x;
}
```

---

5. Balanced Search Trees (BST)

5.1. $2-3$ tree

Allow 1 or 2 keys per node.

- 2-node: 1 key, 2 children
- 3-node: 2 keys, 3 children

Perfect balance. Symmetric order.

5.1.1. Properties of 2-3 tree

Local Transformations. Splitting a 4-node is a **local** transformation: const number of operations.

Invariants. Maintains symmetric order and perfect balance.

Guaranteed **lograthmic** in all kinds of operations.

5.2. Left-learning Red-black BSTs

Worst: $2\ \lg(N)$

Average: $\lg(N)$

- Represent 2-3 tree as a BST.
- Use 'internal' left-leaning links as 'glue' for 3-nodes.

Def. A BST such that:

- No node has 2 red links connected to it.
- Every path from root to null link has the same number of black links
- Red links lean left.

Prop. 1-1 correspondence to 2-3 tree.

5.2.1. Search red-black BSTs

Search is the same as for elementary BST (ignore color), but runs faster because of better balance.

```java
public Val get(Key key)
{
    Node x = root;
    while (x != null)
    {
        int cmp = key.compareTo(x.key);
        if (cmp < 0) x = x.left;
        else if (cmp > 0) x = x.right;
        else return x.val;
    }
    return null;
}
```

5.2.2. Red-black BST Representation

Each node is pointed to by precisely one link (from its parent), so can encode color of lnks in nodes.

```
private static final boolean RED = true;
private static final boolean BLACK = false;

private class Node
{
    Key key;
    Value val;
    Node left, right;
    boolean colr; // color of parent link
}
private boolean isRed(Node x)
{
    if (x == null) return false; // null links are black
    return x.color == RED;
}
```

5.2.3. Red-black BST operations

Left/Right rotation.

Color flip. Recolor to split a temporary 4-node.

Insertion. Keep symmetric and balanced.

- Right child red, left child black: rotate left.
- Left child, left-left grandchild red: rotate right.
- Both children red: flip colors.

```
private Node put(Node h, Key key, Value val)
{
    // insert at bottom and color red
    if (h == null) return new Node(key, val, RED);
    int cmp = key.compareTo(h.key);
    if (cmp < 0) h.left = put(h.left, key, val);
    else if (cmp > 0) h.right = put(h.right, key, val);
    else h.val = val;

    // lean left
    if (isRed(h.right) && !isRed(h.left))  h = rotateLeft(h);
    // balance 4-node
    if (isRed(h.left) && isRead(h.left.left))  h = rotateRight(h);
    // split 4-node
    if (isRead(h.left) && isRead(h.right)) flipColors(h);
    //above: only a few extra lines of code provides near-perfect balance
    return h;
}
```

5.3. B-Trees

Generalize 2-3 trees by allowing up to $M - 1$ [1] key-link pairs per node.

- At least 2 key-link pairs at root.
- At least $M/2$ key-link pairs in other nodes.
- External nodes contain client keys.
- Internal nodes contain copies of keys to guide search.

5.3.1. Balance in B-tree

Prop. A search or an insertion in a B-tree of order M with N keys requires between $\log_{M-1} N$ and $\log_{M/2} N$ probes.

In practice, number of probes is at most 4. For M = 1024, N = 62 million, $\log_{M/2} N \leq 4$

Optimization. Always keep root page in memory.

5.4. Line Segment Intersection Search

Quadratic algorithm: Check all pairs of line segments for intersection.

Nondegeneracy assumption: All x and y-coordinates are distinct.

Sweep vertical line from left to right Algorithm

$O(N \log N)$

- x coordinates define events
- h-segment (left endpoint): insert y-coordinate into BST.
- h-segment (right endpoint): remove y-coordinate from BST.
- v-segment: range search for interval of y-endpoints.

5.5. K dimention-Trees

5.5.1. Space-partitioning trees

Use a `tree` to represent a recursive subdivision of 2d space.

- **Grid.** Divide space uniformly into squares.
- **2d tree**. 2-dimensional orthogonal range search. Recursively divide space into two halfplanes.
- Quadtree. Recursively divide space into 4 quandrants.
- BSP tree. Recursively divide space into 2 regions.

[1] Choose M as large as possible so that M links fit in a page, e.g., M = 1024.

5.5.2. k-dimension Tree

Data Structure. BST, but alternate using x and y coordinates as key.

- Search gives rectangle containing point.
- Insert further subdivides the plane.

Range Search

Typical: $O(\log N)$. Worst: $O(\sqrt{N})$

Goal: Find all points in a query axis-aligned rectangle.

- Check if point in node lies in given rectangle.
- Recursively search left/bottom (if any could fall in rectangle).
- Recursively search right/top (if any could fall in rectangle).

Nearest neighbor search

5.5.3. Kd Tree

Recursively partition k-dimensional space into 2 halfspaces.

Implementation. BST, but cycle through dimensionals ala 2d trees.

Simple data structure for processing k-(high) dimensional data, N-body simulation and clustered data.

5.6. Interval Search Tree

To insert an interval (lo, hi):

- Insert into BST, using lo as the key.
- Update max in each node on search path.

To search for any one interval that intersects query interval (lo, hi):

- If interval in node intersects query interval, return it.
- Else if left subtree is null, go right.
- Else if max endpoint in left subtree is less than lo, go right.
- Else go left.

If search goes **left**, then there is either an intersection in left subtree or no intersections in either.

Implementation. Use red-black BST [2] to guaranterr performance $O(\log N)$.

[2] Easy to maintain auxiliary info using log N extra work per operation.

5.7. Rectangle Intersection

Bootstrapping is not enough if using a quadratic algorithm. Linearithmic algorithm is necessary to sustain Moore's Law.

Sweep-line Algorithm

---

6. Hash Tables

6.1. Hash Tables

Save items in a **key-indexed table**, where index is a function of the key.

Implementation hash code

```java
// Integers
public final class Interger
{
    private final int value;
    private int hashCode()
    {  return value; }
}
// Boolean
public final class Boolean
{
    private final boolean value;
    public int hashCode()
    {
        if (value) return 1231;
        else       return 1237;
    }
}
// Doubles
public final class Double
{
    private final double value;
    public int hashCode()
    {
        long bits = doubleToLongBits(value);
        // convert to IEEE 64-bit representation
        // xor most significant 32-bits with least significant 32-bits
        return (int) (bits ^ (bits >>> 32));
```

```
    }
}
// Strings
public final class String
{
    private final char[] s;
    public int hashCode()
    {
        int hash = 0;
        for (int i = 0; i < length(); i++)
            hash = s[i] + (31 * hash); // s[i] = i-th character of s
        return hash;
    }
}
```

6.2. Separate Chaining

Separate chaining symbol table

Use an array of $M < N$ linked lists.

- Hash: map key to integer i between 0 and $M - 1$.
- Insert: put at front of i^{th} chain, if not already there.
- Search: need to search only i^{th} chain.

Separate Chaining Searching Tree

```
public class SeparateChainingHashST<Key, Value>
{
    private int M = 97; // number of chains
    private Node[] st = new Node[M]; // array of chains
    private static class Node
    {
        private Object key;
        // no generic array creation (declare key and value of type Object)
        private Object val;
        private Node next;
    }
    public Value get(Key key){
        int i = hash(key);
        for (Node x = st[i]; x != null; x = x.next)
            if (key.equals(x.key)) return (Value) x.val;
        return null;
    }
}
```

6.3. Linear Probing

6.3.1. Collision resolution: Open Addressing

When a new key collides, find next empty slot, and put it there.

- Hash. Map key to integer i between 0 and $M-1$.
- Insert. Put at table index i if free; if not try $i+1$, $i+2$, etc.
- Search. Search table index i, if occupied but no match, try $i+1$, $i+2$, etc.
- Note. Array size M must be greater than number of key-value pairs N.

Linear Probing Searching Tree

```
public class LinearProbingHashST<Key, Value>
{
    private int M = 30001;
    private Value[] vals = (Value[]) new Object[M];
    private Key[] keys = (Key[]) new Object[M];
    private int hash(Key key) { /*as before*/}
    public void put(Key key, Value val)
    {
        int i;
        for (i = hash(key); keys[i] != null; i = (i+1)%M)
            if (keys[i].equals(key))
                break;
        keys[i] = key;
        vals[i] = val;
    }
    public Value get(Key key)
    {
        for (int i = hash(key); keys[i] != null; i = (i+1) % M )
            if (key.equals(keys[i]))
                return vals[i];
        return null;
    }
}
```

6.3.2. Hash benefits

- Save time in performing arithmetic.
- Great potential for bad collision patterns.
- For long strings: only examine 8-9 evenly spaced characters.

6.3.3. Separate Chaining vs. Linear Probiing

Separate Chaining.

- Easier to implement delete.
- Performance degrades gracefully.
- Clustering less sensitive to poorly-designed hash function.

Linear Probing.

- Less wasted space.
- Better cache performance.

Two-probe hashing. Double hashing. Cuckoo hashing.

6.3.4. Hash Tables vs. Balanced Search Trees

Hash tables.

- Simpler to code.
- No effective alternative for unordered keys.
- Faster for simple keys (a few arithmetic ops vs. $\log N$ compares.)
- Better system support in Java for strings (e.g., cached hash code)

Balanced search trees.

- Stronger performance guarantee.
- Support for ordered ST operations.
- Easier to implement `compareTo()` correctly than `equals()` and `hashCode()`.

6.4. Searching Tree Summary

6.5. Symbol Table Applications

6.5.1. Exception Filter using Set API

Read in a list of words from a file, print out all words from standard input that are in or not in the list.

ST implementations: summary

implementation	worst-case cost (after N inserts)			average case (after N random inserts)			ordered iteration?	key interface
	search	insert	delete	search hit	insert	delete		
sequential search (unordered list)	N	N	N	N/2	N	N/2	no	equals()
binary search (ordered array)	lg N	N	N	lg N	N/2	N/2	yes	compareTo()
BST	N	N	N	1.38 lg N	1.38 lg N	?	yes	compareTo()
red-black tree	2 lg N	2 lg N	2 lg N	1.00 lg N	1.00 lg N	1.00 lg N	yes	compareTo()
separate chaining	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals() hashCode()
linear probing	lg N *	lg N *	lg N *	3-5 *	3-5 *	3-5 *	no	equals() hashCode()

* under uniform hashing assumption

Figure 2: Searching Tree

```
public class WhiteList
{
    public static void main(String[] args)
    {
        // create empty set of strings
        SET<String> set = new SET<String>();
        // read in whitelist
        In in = new In(args[0]);
        while (!in.isEmpty())
            set.add(in.readString());
        while (!StdIn.isEmpty())
        {
            // print words not in list
            String word = StdIn.readString();
            if (set.contains(word))
                StdOut.println(word);
        }
    }
}
```

6.5.2. Dictionary lookup

Look up DNS; key/values in csv file, etc.

```
public class LookupCSV
{
    public static void main(String[] args)
    { // process input file
        In in = new In(args[0]);
        int keyField = Integer.parseInt(args[1]);
        int valField = Integer.parseInt(args[2]);

        //build symbol table
        ST<String, String> st = new ST<String, String>();
        while (!in.isEmpty())
        {
            String line = in.readLine();
            String[] tokens = line.split(',');
            String key = tokens[keyField];
            String val = tokens[valField];
            st.put(key, val);
        }
        // process lookups with standard I/O
        while (!StdIn.isEmpty())
        {
            String s = StdIn.readString();
            if (!st.contains(s)) StdOut.println('Not found');
            else StdOut.println(st.get(s));
        }
    }
}
```

File Indexing, book index. Concordance.

6.5.3. Sparse Matrix-vector Multiplication

Nested loops: $O(N^2)$

Vector representations. 1D array standard representatin: ($\times$)

- Const time access to elements.
- Space proportional to N.

Symbol table representation: ($\checkmark$)

- Key = index, value = entry.
- Efficient iterator.
- Space proportional to number of non-0s.

Sparse vector data type

```
public class SparseVector
{ // HashST because order not important
    private HashST<Integer, Double> v;

    // empty ST represents all 0s vector
    public SparseVector()
    { v = new HashST<Integer, Double>(); }

    // a[i] = value
    public void put(int i, double x)
    { v.put(i,x); }

    //return a[i]
    public double get(int i)
    {
        if (!v.contains(i)) return 0.0;
        else return v.get(i);
    }
    public Iterable<Integer> indices()
    { return v.keys(); }

    // dot product is const time for sparse vectors
    public double dot(double[] that)
    {
        double sum = 0.0;
        for (int i:indices())
            sum += that[i] * this.get(i);
        return sum;
    }
}
```

Matrix Representations

2D array (standard) matrx representation: Each row of matrix is an **array**.

- Const time access to elements.
- Space proportional to N^2.

Space matrix representation. Each row of matrix is a **sparse vector**.

- Efficient access to elements.
- Space proportinal to number of non-0s (plus N).

---

7. Graph

```
In in = new In(args[0]); // read graph from input stream
Graph G = new Graph(in);

//print out each edge (twice)
for (int v = o0, v < G.V(); v++)
    for (int w: G.adj(v))
        StdOut.println(v + '-' + w);
```

Compute degree of v

```
public static int degree(Graph G, int v)
{
    int degree = 0;
    for (int w : G.adj(v)) degree ++;
    return degree;
}
```

Compute max degree

```
public static int maxDegree(Graph G)
{
    int max = 0;
    for (int v = 0; v < G.V(); v++)
        if (degree(G,v) > max)
            max = degree(G,v);
    return max;
}
```

Compute average degree

```
public static double averageDegree(Graph G)
{ return 2.0 * G.E() / G.V(); }
```

Count self-loops

```java
public static int numberOfSelfLoops(Graph G)
{
    int count = 0;
    for (int v = 0; v < G.V(); v++)
        for (int w : G.adj(v))
            if (v == m) count ++;
    return count/2;
}
```

Adjacency-list graph.

7.1. Depth-First Search (DFS)

Maze search: explore every intersection in the maze.

- Vertex = intersection.
- Edge = passage.

DFS. Symmetrically search through a graph. Mimic maze exploration.

Applications.

- Find all vertices connected to a given source vertex.
- Find a path between 2 vertices.

Depth-first search

```java
public class DepthFirstPaths
{
    private boolean[] marked; // marked[v] = true if v connected to s
    private int[] edgeTo; // edgeTo[v] = previous vertex n path from s to v
    private int s;

    // initialize data structures, find vertices connected to s
    public DepthFirstPaths(Graph G, int s)
    {
        /// omitted...
        dfs(G,s);
    }
    //recursive DFS does the work
    private void dfs(Graph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
```

```
            if (!marked[w])
            {
                dfs(G,w);
                edgeTo[w] = v;
            }
    }
}
```

Prop. DFS marks all vertices connected to s in time proportion to the sum of their degrees. After DFS, can find vertices connected to s in const time and can find a path to s in time proportional to its length.

7.1.1. DFS Application

Flood fill in photography.

7.2. Breadth-First Search (BFS)

Put s onto a FIFO queue, and mark s as visited.

Repeat until queue is empty:

- Remove vertex v from queue.
- Add to queue all unmarked vertices adjacent neighbors to v and mark them as visited.

Prop. BFS computes shortest paths from s to all other vertices in a graph in time proportional to $E + V$.

BFS

```
public class BreadthFirstPaths
{
    private boolean[] marked;
    private int[] edgeTo;
    private void bfs(Graph G, int s)
    {
        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        while (!q.isEmpty())
        {
            int v = q.dequeue();
            for (int w : G.adj(v))
```

```
                {
                    if (!marked[w])
                    {
                        q.enqueue(w);
                        marked[w] = true;
                        edgeTo[w] = v;
                    }
                }
            }
        }
}
```

7.2.1. Comparison

DFS. Put unvisited vertices on a **stack**.

BFS. Put unvisited vertices on a **queue**.

Shortest path. Find path from s to t that uses **fewest number of edges**.

7.2.2. BFS Application

Routing. Shortest path.

7.3. Connected Components

Is v connected to w in const time? Union find $\times$, DFS ✓.

Finding connected components with DFS.

```
public class CC
{
    private boolean marked[];
    private int[] id; // id[v] = id of component containing v
    private int count; // number of components

    public CC(Graph G)
    {
        marked = new boolean[G.V()];
        id = new int[G.V()];
        for (int v = 0; v < G.V(); v++)
        {
            if (!marked[v])
            {
                // run DFS from one vertex in each components
```

```
                dfs(G,v);
                count++
            }
        }
    }
    public int count()
    { return count; } // number of components
    public int id(int v)
    { return id[v];} // id of component containing v
    private vooid dfs(Graph G, int v)
    {
        // all vertices discovered in same call of dfs have same id
        marked[v] = true;
        id[v] = count;
        for (int w:G.adj(v))
            if (!marked[w])
                dfs(G,w);
    }
}
```

7.4. Directed Graph

7.4.1. DFS in digraph

DFS is a **diagraph** algorithm. BFS is a **digraph** algorithm. BFS computes shortest paths.

DFS for directed graph.

```
public class DirectdDFS
{
    private boolean[] marked; // true if path from s
    public DirectedDFS(Digraph G, int s)
    { // constructor marks vertices reachable from s
        marked = new boolean[G.V()];
        dfs(G,s);
    }
    // recursive DFS does the work
    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G,w);
    }
    // client can ask whether any vertex is reachable from s
```

```
    public boolean visited(int v)
    {  return marked[v];  }
}
```

7.5. Topological Sort

- Run depth-first search.
- Return vertices n reverse postorder.

```
public class DepthFirstOrder
{
    private boolean[] marked;
    private Stack<Integer> reversePost;

    public DepthFirstOrder(Digraph G)
    {
        reversePost = new Stack<Integer>();
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            if (!marked[v]) dfs(G,v);
    }
    private void dfs(Digraph G, int v)
    {
        marked[v] = true;
        for (int w : G.adj(v))
            if (!marked[w]) dfs(G,w);
        reversePost.push(v);
    }
    // return all vertices in reverse DFS postorder
    public Iterable<Integer> reversePost()
    { return reversePost; }
}
```

Prop. Reverse DFS postorder of a DAG is a topological order.

7.6. Strong Components

If there is a directedd path from v to w **and** a directed path from w to v.

Two DFS is needed. Only one line change in the code.

```
        DepthFirstOrder dfs = new DepthFirstOrder(G.reverse());
        for (int v : dfs.reversePost())
```

8. Minimum Spanning Trees (MST)

Given. Undirected graph G with positive edge weights connected.

Def. A **spanning tree** of G is a subgraph T that is both a **tree** (connected, acyclic) and **spanning** (includes all of the vertices).

Goal. Find a min weight spanning tree.

8.1. Greedy MST Algorithm

Def. A **cut** in a graph is a partition of its vertices into 2 sets.

Def. A **crossing edge** connects a vertex in one set with a vertex in the order.

Prop. Given any cut, the crossing edge of min weight is in the MST.

Greedy MST algorithm: efficient implementations. Choose cut? Find min-weight edge?

- Kruskal's algorithm.
- Prim's algorithm.

8.2. Kruskal's Algorithm

Consider edges in ascending order of weight. Add next edge to tree T unless doing so would create a cycle.

Challenge. Would adding edge $v - w$ to tree T create a cycle? If not, add it.

Efficient solution. Use the **union-find** data structure.

- Maintain a set for each connected component in T.
- If v and w are in same set, then adding $v - w$ would create a cycle.
- To add $v - w$ to T, merge sets containing v and w.

Kruskal's Algorithm

Worst: $O(E \log E)$

```
public class KruskalMST
{
    private Queue<Edge> mst = new Queue<Edge>();
    //build priority queue
    public KruskalMST(EdgeWeightedGraph G)
    {
```

```
        MinPQ<Edge> pq = new MinPQ<Edge>();
        for (Edge e : G.edges())
            pq.insert(e);
        UF uf = new UF(G.V());
        while (!pq.isEmpty() && mst.size() < G.V() - 1)
        {
            // greedily add edges to MST
            Edge e = pd.delMin();
            int v = e.either(), w = e.other(v);
            //edge v-w doesn't create cycle
            if (!uf.connected(v,w))
            {
                // merge sets
                uf.union(v,w);
                // add edge to MST
                mst.enqueue(e);
            }
        }
    }
    public Iterable<Edge> edges()
    { return mst; }
}
```

8.3. Prim's Algorithm

Idea

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V - 1$ edges.

Challenge. Find the min weight edge with exactly one endpoint in T.

Lazy solution. Maintain a PQ of **edges** with at least one endpoint in T.

- Key = edge; priority = weight of edge.
- Delete-min to determine next edge $e = v - w$ to add to T.
- Disregard if both endpoints v and w are in T.
- Otherwise, let w be the vertex not in T:
  - add to PQ any edge incident to w (assuming other endpoint not in T)
  - add w to T

8.3.1. Prim's Algorithm: lazy implementation

Worst: $O(E \log E)$

```
public class LazyPrimMST
{
    private boolean[] marked; // MST vertices
    private Queue<Edge> mst; // MST edges
    private MinPQ<Edge> mq; // PQ of edges

    public LazyPrimMST(WeightedGraph G)
    {
        pq = new MinPQ<Edge>();
        mst = new Queue<Edge>();
        marked = new boolean[G.V()];
        // assume G is connected
        visit(G,0);

        //repeatedly delete the min weight edge e = v-w from PQ
        while (!pq.isEmpty())
        {
            Edge e = pd.delMin();
            int v = e.either(), w = e.other(v);
            //ignore if both endpoints in T add edge e to tree
            if (marked[v] && marked[w]) continue;
            //add edge e to tree
            mst.enqueue(e);
            //add v or w to tree
            if (!marked[v]) visit(G,v);
            if (!marked[w]) visit(G,w;)
        }
    }
    private void visit(WeightedGraph G, int v)
    {
        // add v to T
        marked[v] = true;
        for (Edge e: G.adj(v))
        // for each edge e = v-w, add to PQ if w not already in T
            if (!marked[e.other(v)])
                pq.insert(e);
    }
    public Iterable<Edge> mst()
    { return mst; }
}
```

8.3.2. Prim's algorithm: eager implementation

Challenge. Find min weight edge with exactly one endpoint in T.

Eager solution. Maintain a PQ of **vertices** connected by an edge to T, where priority of vertex v = weight of shortese edge connecting v to T.

Idea

- Start with vertex 0 and greedily grow tree T.
- Add to T the min weight edge with exactly one endpoint in T.
- Repeat until $V-1$ edges.

8.3.3. Application

Euclidean MST. k-clustering.

8.4. Dijkstra Algorithm

Disjstra's algorithm computes a SPT in any edge-weighted digraph with nonnegative weights.

- Consider vertices in increasing order of distance from s.
- Add vertex to tree and relax all edges pointing from that vertex.

```
public class DijkstraSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;
    private IndexMinPQ<Double> pq;

    public DijkstraSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];
        pq = new IndexMinPQ<Double>(G.V());

        for (int v = 0; v < G.V(); v++)
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0;

        //relax vertices in order of distance from s
        pq.insert(s,0.0);
        while (!pq.isEmpty())
```

```
        {
            int v = pd.delMin();
            for (DirectedEdge e : G.adj(v))
                relax(e);
        }
    }
    private void relax(DirectedEdge e)
    {
        int v = e.from(), w = e.to();
        if (distTo[w] > distTo[v] + e.weight())
        {
            distTo[w] = distTo[v] + e.weight();
            edgeTo[w] = e;
            //update PQ
            if (pq.contains(w)) pd.decreaseKey(w,distTo[w]);
            else pq.insert (w,distTo[w]);
        }
    }
}
```

8.4.1. Computing Spanning Trees in graphs

- Prim's algorithm is essentially the **same** as Dijkastra's algorithm.
- Both are in a family of algorithms that compute a graph's **spanning tree**. (DFS and BFS are also in this family of algorithms)

Difference: rule used to choose next vertex for the tree.

- Prim's: Closest vertex to the *tree* (via an undirected edge).
- Dijkstra's: Closest vertex to the *source* (via a directed path).

8.5. Shortest paths in Edge-weighted DAGs

```
public class AcyclicSP
{
    private DirectedEdge[] edgeTo;
    private double[] distTo;

    public AcyclicSP(EdgeWeightedDigraph G, int s)
    {
        edgeTo = new DirectedEdge[G.V()];
        distTo = new double[G.V()];

        for (int v = 0; v < G.V(); v++)
```

```
            distTo[v] = Double.POSITIVE_INFINITY;
        distTo[s] = 0.0
        // topological order
        Topological topological = new Topological(G);
        for (int v : topological.order())
            for (DirectedEdge e: G.adj(v))
                relax(e);
    }
}
```

8.5.1. Application

Seam carving (resizing image without distortion).

A SPT exists iff no negative cycles.

Bellman-Ford algorithm. Dynamic programming algorithm computes SPT.

8.5.2. Shortest Path Summary

Dijkstra's algorithm

- Nearly linear-time when weights are nonnegative.
- Generalization encompasses DFS, BFS, and Prim.

Acyclic edge-weighted digraphs.

- Arise in applications.
- Faster than Dijkstra's algorithm.
- Negative weights are no problem.

Negative weights and negative cycles.

- Arise in applications.
- If no negative cycles, can find shortest path via Bellman-Ford.
- If negative cycles, can find one via Bellman-Ford.

---

9. Max Flow & Min Cut, Radix Sort

Max Flow and Min Cut are dual problems.

9.1. Mincut

Input. An edge-weighted digraph, souce vertex s, target vertex t.

Def. An $st-$**cut** is a partition of the vertices into 2 disjoint sets, with s in one set A and t in the other set B.

Def. Its **capacity** is the sum of the capacities of the edges from A to B.

Minimum st-cut (mincut) problem. Find a cut of minimum capacity.

9.2. Maxflow

Def. An $st-$**flow** is an assignment of values to the edges such that:

- Capacity constraint: $0 \leq$ edge's flow $\leq$ edge's capacity.
- Local equilibrium: inflow = outflow at every vertex (except s and t).

Def. The **value** of a flow is the inflow at t.

Maximum st-flow (maxflow) problem. Find a flow of a maximum value.

9.3. Ford-Fulkerson algorithm

Idea. Increase flow along augmenting paths.

Augmenting path. Find an undirected path from s to t such that:

- Can increase flow on forward edges (not full).
- Can decrease flow on backward edge (not empty).

9.4. Maxflow-Mincut Theorem

Def. The **net flow across** a cut (A, B) is the sum of the flows on its edges from A to B minus the sum of the flows on its edges from B to A.

Flow-value lemma. Let f be any flow and let (A,B) be any cut. Then, the net flow across (A,B) equals the value of f.

9.4.1. Relationship between flows and cuts

Weak duality. Let f be any flow and let (A, B) be any cut. Then, the value of the flow $\leq$ the capacity of the cut.

9.4.2. Augmenting Path Theorem.

A flow f is a maxflow iff no augmenting paths.

9.4.3. Maxflow-mincut Theorem.

Value of the maxflow = capacity of mincut.

9.4.4. Compute mincut (A, B) from maxflow f

- By augmenting path theorem, no augmenting paths w.r.t. f.
- Compare A = set of vertices connected to s by an undirected path with no full forward or empty backward edges.

1. Shortest path: augmenting path with fewest number of edges.
2. Fasttest path: augmenting path with maximum bottleneck capacity.

9.4.5. Flow network representation

```
public class FlowEdge
{
    private final int v, w; // from and to
    private final double capacity; // capacity
    private double flow; // flow

    public FlowEdge(int v, int w, double capacity)
    {
        this.v = v;
        this.w = w;
        this.capacity = capacity;
    }
    public int from() { return v;}
    public int to() { return w; }
    public double capacity() { return capacity; }
    public double flow() { return flow;}
    public int other(int vertex)
    {
        if (vertex == v) return w;
        else if (vertex == w) return v;
    }

    public double residualCapacityTo(int vertex)
    {
```

```
        if (vertex == v) return flow; // backward edge
        else if (vertex == w) return capacity - flow; //forward edge
    }
    public void addResidualFlowTo(int vertex, double delta)
    {
        if (vertex == v) flow -= delta; //backward edge
        else if (vertex == w) flow += delta; // forward edge
    }
}
```

Flow network.

```
public class FlowNetwork
{
    private fnal int V;
    // same as EdgeWeightedGraph, but adjacency lists of FlowEdges instead of Edges
    private Bag<FlowEdge>[] adj;
    public FlowNetwork(int V)
    {
        this.V = V;
        adj = (Bag<FlowEdge>[]) new Bag[V];
        for (int v = 0; v < V; v++)
            adj[v] = new Bag<FlowEdge>();
    }
    public void addEdge(FlowEdge e)
    {
        int v = e.from();
        int w = e.to();
        // add forward edge
        adj[v].add(e);
        // add backward edge
        adj[w].add(e);
    }
    public Iterable<FlowEdge> adj(int v)
    { return adj[v]; }
}
```

Ford-Fulkerson

```
public class ForFulkerson
{
    private boolean[] marked; // true if s->v path in residual network
    private FlowEdge[] edgeTo; // last edge on s->v path
    private double value; // value of flow
    public FordFulkerson(FlowNetwork G, int s, int t)
```

```
    {
        value = 0.0;
        while (hasAugmentingPath(G,s,t))
        {
            double bottle = Double.POSITIVE_INFINITY;
            // compute bottleneck capacity
            for (int v = t; v != s; v = edgeTo[v].other(v))
                bottle = Math.min(bottle, edgeTo[v].residualCapacityTo(v));
            //augment flow
            for (int v = t; v != s; v = edgeTo[v].other(v))
                edgeTo[v].addResidualFlowTo(v,bottle);
            value += bottle;
        }
    }
    private boolean hasAugmentingPath(FlowNetwork G, int s, int t)
    {
        edgeTo = new FlowEdge[G.V()];
        marked = new boolean[G.V()];

        Queue<Integer> q = new Queue<Integer>();
        q.enqueue(s);
        marked[s] = true;
        while (!q.isEmpty())
        {
            int v = q.dequeue();
            for (FlowEdge e : G.adj(v))
            {
                int w = e.other(v);
                // found path from s to w in the residual network?
                if (e.residualCapacityTo(w) > 0 && !marked[w])
                {
                    edgeTo[w] = e; // save last edge on path to w
                    marked[w] = true; // mark w
                    q.enqueue(w); // add w to the queue
                }
            }
        }
    }
    public double value()
    { return value;}
    public boolean inCut(int v)
    // is v reachable from s in residual network?
    { return marked[v]; }
    return marked[t]; // is t reachable from s in residual network?
}
```

9.4.6. Application

Bipartitie matching problem.

9.5. Mincut

Mincut (A, B).

- Let S = students on s side of cut.
- Let T = companies on s side of cut.
- Fact. $|S| > |T|$; students in S can be matched only to companies in T.

9.6. Summary on Max Flow and Min Cut

Mincut problem. Find an *st*-cut of min capacity.

Maxflow problem. Find an *st*-flow of max value.

Duality. Value of the maxflow = capacity of mincut.

9.7. Radix Sort

How to efficiently reverse a string?

Quadratic time.

```java
public static String reverse(String s)
{
    String rev = "";
    for (int i = s.length() - 1; i >= 0; i--)
        rev += s.charAt(i);
    return rev;
}
```

Linear time.

```java
public static String reverse(String s)
{
    StringBuilder rev = new StringBuilder();
    for (int i = s.length() - 1; i >= 0; i--)
        rev.append(s.charAt(i));
    return rev.toString();
}
```

9.8. LSD Radix (String) Sort

- Consider characters from right to left.
- Stably sort using d^{th} character as the key (using key-indexed counting).

9.9. MSD String Sort

Disadvantages of MSD string sort.

- Access memory ‘randomly’ (cache inefficient).
- Inner loop has a lot of instructions.
- Extra space for `count[]`.
- Extra space for `aux[]`.

Disadvantage of quicksort.

- Linearithmic number of string compares (not linear).
- Has to rescan many characters in keys with long prefix matches.

9.10. 3-way string quicksort

Do 3-way partitioning on the d^{th} character.

- Less overhead than R-way partitioning in MSD string sort.
- Doesn’t re-examine characters equal to the partitioning char (but does re-examine characters not equal to the partitioning char).

```
private static void sort(String[] a)
{ sort(a, 0, a.length - 1, o); }

private static voi sort(String[] a, int lo, int hi, int d)
{
    // 3-way partitioning using d-th character
    if (hi <= lo) return;
    int lt = lo, gt = hi;
    int v = charAt(a[lo], d); // charAt to hanle variable-length strings
    int i = lo + 1;
    while (i <= gt)
    {
        int t = charAt(a[i], d);
        if (t < v) exch(a,lt++, i++);
        else if (t > v) exch(a,i,gt--);
```

```
            else i++;
        }
        sort(a, lo, lt-1, d);
        if (v>=0) sort(a,lt,gt,d+1); // sort 3 subarrays recursively
        sort(a, gt+1, hi, d);
    }
```

9.10.1. Comparison: 3-way string quicksort vs. standard quicksort

Standard Quicksort.

- Uses $2N \ln N$ **string compares** on average.
- Costly for keys with long common prefixes. (quite common)

3-way string (radix) quicksort

- Uses $2N \ln N$ **character compares** on average for random strings.
- Avoids re-comparing long common prefixes.

9.10.2. Comparison: 3-way string quicksort vs. MSD string sort

MSD string sort.

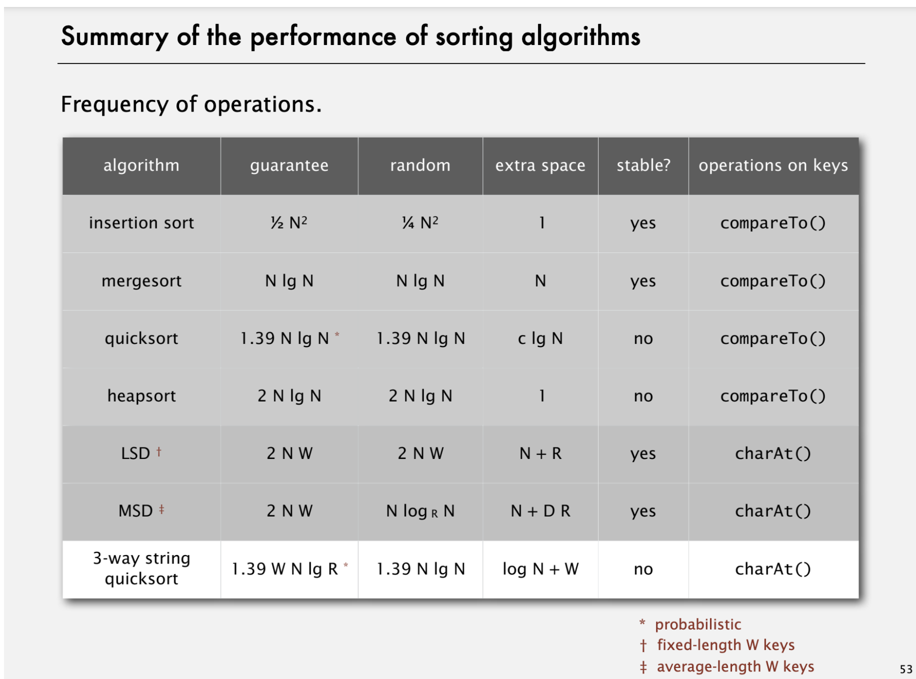
- Cache-inefficient.
- Too much memory storing `count[]`.
- Too much overhead reinitializing `count[]` and `aux[]`.

3-way string quicksort.

- Cache-friendly.
- Has a short inner loop.
- Is in-place.

9.11. Summary for Sorting Algorithms

- D = function-call stack depth (length of longest prefix match)



Summary of the performance of sorting algorithms

Frequency of operations.

algorithm	guarantee	random	extra space	stable?	operations on keys
insertion sort	½ N^2	¼ N^2	1	yes	compareTo()
mergesort	N lg N	N lg N	N	yes	compareTo()
quicksort	1.39 N lg N *	1.39 N lg N	c lg N	no	compareTo()
heapsort	2 N lg N	2 N lg N	1	no	compareTo()
LSD †	2 N W	2 N W	N + R	yes	charAt()
MSD ‡	2 N W	N $\log_R$ N	N + D R	yes	charAt()
3-way string quicksort	1.39 W N lg R *	1.39 N lg N	log N + W	no	charAt()

* probabilistic
† fixed-length W keys
‡ average-length W keys

53

Figure 3: Sorting Algorithms

10. Tries and Substring Search

10.1. R-way Tries

String symbol-table. Symbol table specialized to string keys.

Recall String symbol table: Red-black BST, hashing (linear probing).

10.1.1. Tries

Come from re-**trie**-val. [Pronounced "try"]

- Store *characters* in nodes (not keys).
- Each node has R children, one for each possible character.
- Store values in nodes corresponding to last characters in keys.

10.1.2. Search in a trie

- Search hit: node where search ends has a non-null value.
- Search miss: reach null link or node where search ends has null value.

10.1.3. Insertion into a trie

- Encounter a null link: create a new node.
- Encounter the last character of the key: set value in that node.

10.1.4. Implementation

Node. A value, plus references to R nodes.

R-way trie.

```
public class TrieST<Value>
{
    private static final int R = 256; // extended ASCII
    private Node root = new Node();

    private static class Node
    {
    // use 'Object' instead of Value since no generic array creation in Java
    private Object value;
    private Node[] next = new Node[R];
    }
```

```java
    public void put(String key, Value val)
    { root = put(root, key, val, 0); }

    private Node put(Node x, String key, Value val, int d)
    {
        if (x == null) x = new Node();
        if (d == key.length()) {x.val = val; return x; }
        char c = key.charAt(d);
        x.next[c] = put(x.next[c], key, val, d+1);
        return x;
    }
    public boolean contains(String key)
    { return get(key) != null; }
    public Value get(String key)
    {
        Node x = get(root, key, 0);
        if (x == null) return null;
        return (Value) x.val; // cast needed
    }

    private Node get(Node x, String key, int d)
    {
        if (x == null) return null;
        if (d == key.length()) return x;
        char c = key.charAt(d);
        return get(x.next[c], key, d+1);
    }
}
```

10.1.5. Trie Performance

Search hit. Needd to examine all L characters for equality.

Search miss.

- Could have mismatch on first character.
- Typical case: examine only a few characters (sublinear).

Space. R null links at each leaf. (But sublinear space possible if many short strings share common prefixes)

Fast search hit and even faster search miss, but wastes space.

10.2. Ternary Search Tries (TST)

- Store characters and values in nodes (not keys).
- Each node has 3 children: smaller left, equal middle, larger right.

10.2.1. Search in TST

Follow links corresponding to each character in the key. If less, take left link; if greater, take right link. If equal, take the middle link and move to the next key character.

Search hit. Node where search ends has a non-null value.

Search miss. Reach a null link or node where search ends has null value.

10.2.2. Comparison: 26-way trie vs. TST

26-way trie. 26 null links in each leaf.

TST. 3 null links in each leaf.

10.2.3. TST implementaton

A TST node is 5 fields: value, character c, reference to left TST, reference to middle TST, reference to right TST.

```
public class TST<Value>
{
    private Node root;
    private class Node
    { /**/}
    public void put(String key, Value val)
    { root = put(root,key,val,0);}
    private Node put(Node x, String key, Value val, int d)
    {
        char c = key.charAt(d);
        if (x == null) { x = new Node(); x.c = c;}
        if (c < x.c) x.left = put(x.left, key, val, d);
        else if (c > x.c) x.right = put(x.right, key, val, d);
        else if (d < key.length() - 1) x.mid = put(x.mid, key, val, d+1);
        else x.val = val;
        return x;
    }
}
```

10.2.4. Comparison: TST vs. Hashing

Hashing.

- Need to examine entire key.
- Search hits and misses cost about the same.
- Performance relies on hash function.
- Doesn't support ordered symbol table operations.

TSTs.

- Works only for strings (or digital keys).
- Only examines just enough key characters.
- Search miss may involve only a few characters.
- Supports ordered symbol table operations.
- Faster than hashing (especially for search misses)
- More flexible than red-black BSTs.

10.2.5. Longest Prefix in an R-way trie

Find longest key in symbol table that is a prefix of query string.

- Search for query string.
- Keep track of longest key encountered.

```
public String longestPrefixOf(String query)
{
    int length = search(root, query, 0, 0);
    return query.substring(0, length);
}
private int search(Node x, String query, int d, int length)
{
    if (x == null) return length;
    if (x.val != null) length = d;
    if (d == query.length()) return length;
    char c = query.charAt(d);
    return search(x.nect[c], query, d+1, length);
}
```

10.3. Suffix Tree

- Patricia trie of suffixes of a string
- Linear-time construction.

10.4. Summary on String symbol tables

10.4.1. Red-black BST

- Performance guarantee: $\log N$ key compares.
- Supports ordered symbol table API.

10.4.2. Hash tables

- Performance guarantee: const number of probes.
- Requires good hash function for key type.

10.4.3. Tries. R-way, TST.

- Performance guarantee: $\log N$ **characters** assessed.
- Supports character-based operations.

10.5. Substring Search

Goal: Find pattern of length M in a text of length N.

Screen scraping: using `indexOf()` method to return the index of the first occurence of a given string.

```
public class StockQuote
{
    public static void main(String[] args)
    {
        String name = 'http://finance.yahoo.com/q?s=';
        In in = new In(name+args[0]);
        String text = in.readAll();
        int start = text.indexOf('Last Trade:', 0);
        int from = text.indexOf('<b>', start);
        int to = text.indexOf('</b>',from);
        String price = text.substring(from + 3, to);
        StdOut.println(price);
    }
}
```

10.6. Brute-force substring search

Check for pattern starting at each text position.

```java
public static int search(String pat, String txt)
{
    int M = pat.length();
    int N = txt.length();
    for (int i = 0; i<= N-M; i++)
    {
        int j;
        for (j = 0; j < M; j++)
            if (txt.charAt(i+j) != pat.charAt(j))
                break;
        //index in text where pattern starts
        if (j == M) return o;
    }
    return N; // not found
}
```

10.7. Knuth-Morris-Pratt (KMP)

Mismatch transition. For each state `j` and `char c != pt.charAt(j)`, set `dfa[c][j] = dfa[c][x]`; then update `X = dfa[pat.charAt(j)][X]`.

10.7.1. Constructing DFA for KMP substring search

For each state `j`:

- Copy `dfa[][X]` to `dfa[][j]` for mismatch case.
- Set `dfa[pat.charAt(j)][j]` to `j+1` for match case.
- Update X.

```java
public KMP(String pat)
{
    this.pat = pat;
    M = pat.length();
    dfa = new int[R][M];
    dfa[pat.charAt(0)][0] = 1;
    for (int X = 0, j = 1; j < M; j++)
    {
        for (int c = 0; c < R; c++)
        // copy mismatch cases
            dfa[c][j] = dfa[c][X];
        // set match case
        dfa[pat.charAt(j)][j] = j + 1;
        // update restart state
        X = dfa[pat.charAt(j)][X];
```

```
    }
}
```

Running time. M character accesses. KMP constructs `dfa[][]` in time and space proportional to RM).

10.7.2. KMP substring search analysis

Prop. KMP substring search accesses no more than $M + N$ chars to search for a pattern of length M in a text of length N.

10.7.3. Rabin-Karp

Monte Carlo version: return match is hash match. Always linear in time.

```java
public int search(String txt)
{
    // check for hash collision using rolling hash function
    int N = txt.length();
    int txtHash = hash(txt, M);
    if (patHash == txtHash) return 0;
    for (int i = M; i < N; i++)
    {
        txtHash = (txtHash + Q - RM*txt.charAt(i-M) % Q) %Q;
        txtHash = (txtHash*R + txt.charAt(i)) % Q;
        // Las Vegas version: check for substring match if hash match;
        // continue search if false collision
        if (patHash == txtHash) return i - M + 1;
    }
    return N;
}
```

---

11. Regular Expressions and Data Compression

11.1. Regular Expression

Pattern Matching.

Substring search. Find a single string in text.

Pattern matching. Find one of a **specified set** of strings in text.

A **regular expression** is a notation to specify a set of strings.

Grep.

Validity checking. Does the `input` match the `regexp`? Use java string library `input.matches(regexp)` for basic RE matching.

```
public class Validate
{
    public static void main(String[] args)
    {
        String regexp = args[0];
        String input = args[1];
        StdOut.println(input.matches(regexp));
    }
}
```

11.2. Data Compression

Compression reduces the size of a file: to save space when storing it, to save time when transmitting it, and most files have lots of redundancy.

11.2.1. Lossless compression and expansion

Binary data B we want to compress. Generate a *compressed* representation $C(B)$, and reconstructs original bitstream B.

11.3. Huffman Coding

Use different number of bits to encode different chars. Ex: Morse code.

11.3.1. Variable-length codes

How do we avoid ambiguity? Ensure that no codeword is a **prefix** of another.

11.3.2. Prefix-tree codes

Trie representation

How to represent the prefix-free code?

A binary trie.

- Chars in leaves.

- Codeword is path from root to leaf.

Compression

1. Start at leaf; follow path up to the root; print bits in reverse.
2. Or: Create ST of key-value pairs.

Expansion

- Start at root.
- Go left if bits = 0; go right if = 1
- If leaf node, print char and return to root.

Huffman trie node data type

```
private static class Node implements Comparable<Node>
{
    private char ch; // unused for internal nodes
    private int freq; // unused for expand
    private final Node left, right;

    public Node(char ch, int freq, Node left, Node right)
    {
        // initializing constructor
        this.ch = ch;
        this.freq = freq;
        this.left = left;
        this.right = right;
    }
    // is Node a leaf?
    public boolean isLeaf()
    { return left == null && right == null; }
    // compare Nodes by frequency
    public int compareTo(Node that)
    { return this.freq - that.freq; }
}
```

Prefix-free codes: expansion

```
public void expand()
{
    // read in encoding trie
    Node root = readTrie();
```

```java
    // read in number of chars
    int N = BinaryStdIn.readInt();

    for (int i = 0; i < N; i++)
    {
        // expand codeword for i-th char
        Node x = root;
        while (!x.isLeaf())
        {
            if (!BinaryStdIn.readBoolean())
                x = x.left;
            else
                x = x.right;
        }
        BinaryStdOut.write(x.ch, 8);
    }
    BinaryStdOut.close();
}
```

How to write the trie?

Write preorder traversal of trie; mark leaf and internal nodes with a bit.

11.3.3. Huffman Algorithm

- Select 2 tries wiith min weight.
- Merge into single trie with cumulative weight.

11.3.4. Huffman codes

How to find best prefix-tree code?

Huffman algorithm.

- Count frequency `freq[i]` for each char `i` in input.
- Start with 1 node correspondin to each char `i` (with weight `freq[i]`)
- Repeat until single trie formed:
  - select 2 tries with min weight `freq[i]` and `freq[j]`
  - merge into single trie with weight `freq[i] + freq[j]`

11.3.5. Constructing a Huffman encoding trie

```java
private static Node buildTrie(int[] freq)
{
```

```
    MinPQ<Node> pq = new MinPQ<Node>();
    // initialize PQ with singleton tries
    for (char i = 0; i < R; i++)
        if (freq[i] > 0)
            pq.insert(new Node(i, freq[i], null, null));
    // merge 2 smallest tries
    while (pq.size() > 1)
    {
        Node x = pq.delMin();
        Node y = pd.delMin();
        // '\0': not used for internal nodes
        // x.freq + y.freq = total frequency
        // x, y: 2 subtries
        Node parent = new Node('\0', x.freq + y.freq, x, y);
    }
    return pq.delMin();
}
```

11.3.6. Summary: Huffman encoding

Huffman algorithm produces an **optimal** [3] prefix-tree code.

Implementation

1. Tabulate char frequencies and build trie.
2. Encode file by traversing trie or lookup table.

Running time. Using a binary heap $\Rightarrow N + R \log R$ (input size + alphabet size).

11.4. Statistical methods comparison

Static model. Same model for all texts.

- Fast
- Not optimal: different texts have different statistical properties.
- *Ex*: ASCII, Morse code

Dynamic model. Generate model based on text.

- Preliminary pass needed to generate model

[3] no prefix-free code uses fewer bits

- Must transmit the model
- *Ex*: Huffman code

Adaptive model. Progressively learn and update model as you read text.

- More accurate modeling produces better compression.
- Decoding must start from beginning.
- *Ex*: Lempel-Ziv-Welch (LZW) Compression.

11.5. Data Compression Summary

Lossless compression.

- Represent fixed-length symbols with variable-length codes. [Huffman]
- Represent variable-length symbols with fixed-length codes. [LZW]

Theoretical limits on compression.

Shannon entropy: $H(X) = -\Sigma_i^n p(x_i) \lg p(x_i)$

---

12. Reductions, Linear Programming and Intractability

12.1. Reductions

Def. Problem X **reduces to** problem Y if you can use an algorithm that solves Y to help solve X.

Cost of solving X = total cost of solving Y + cost of reduction

Linear-time reductions (*Figure 4*)

12.2. NP-Completeness

Poly-time reductions from boolean satisfiability (*Figure 5*)

P. Class of search problems solvable in poly-time.

NP. Class of all search problems, some of which seem wickedly hard.

NP-complete. Hardest problems in NP. Including many fundamental problems: SAT, ILP, 3-COLOR, 3D-ISING, Hamilton Path.

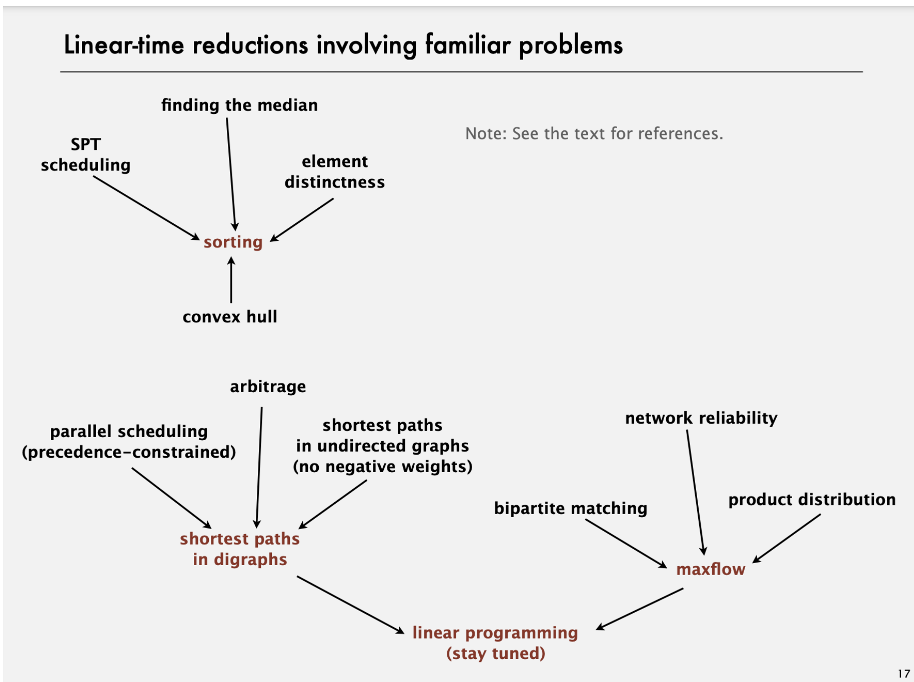


Figure 4: Linear-time reductions

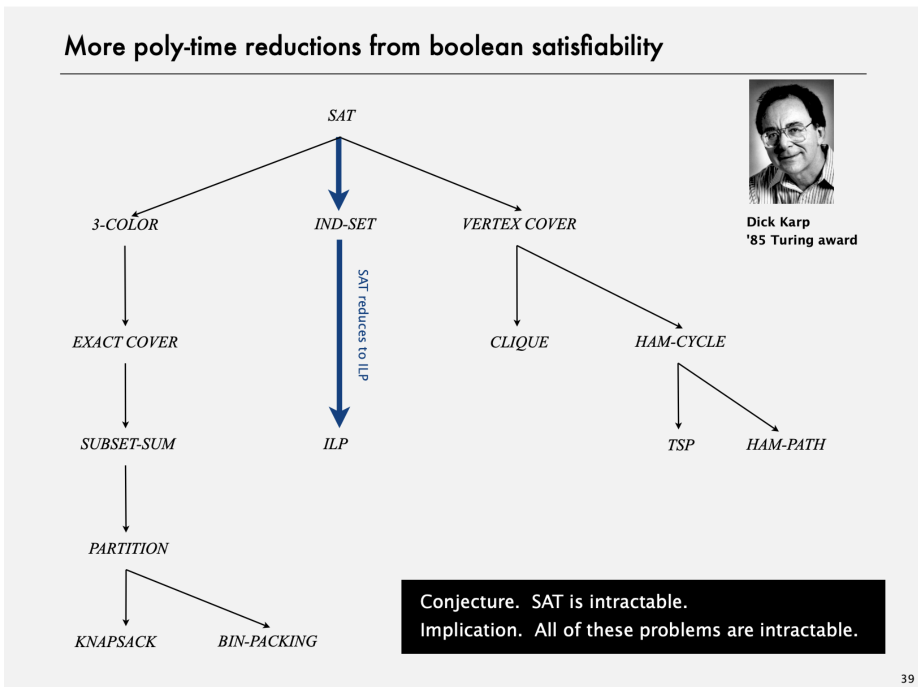


Figure 5: Poly-time reductions from boolean satisfiability

Intractable. Problem with no poly-time algorithm

Poly-time reductions from boolean satisfiability

Hamilton path.

```
public class HamiltonPath
{
    private boolean[] marked; // vertices on current path
    private int count = 0; // number of Hamilton paths
    public HamiltonPath(Graph G)
    {
        marked = new boolean[G.V()];
        for (int v = 0; v < G.V(); v++)
            dfs(G, v, 1);
    }
    private void dfs(Graph G, int v, int depth)
    // depth is length of current path = depth of recursion
    {
        marked[v] = true;
        if (depth == G.V()) count ++; // found one
        for (int w : G.adj(v))
        // backtrack if w is already part of path
            if (!marked[w]) dfs(G, w, depth+1);
        marked[v] = false; // clean up
    }
}
```

$\mathbb{E}$nd - $\mathbb{O}$f - $\mathbb{T}$he - $\mathbb{C}$ourse - $\mathbb{N}$otes