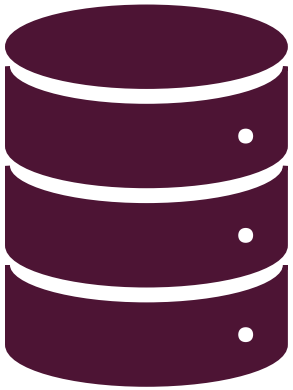


DATA STRUCTURES

WENYE LI
CUHK-SZ

OUTLINE

- Sorting
  - Simple Sorting
  - Advanced Sorting
- Hashing

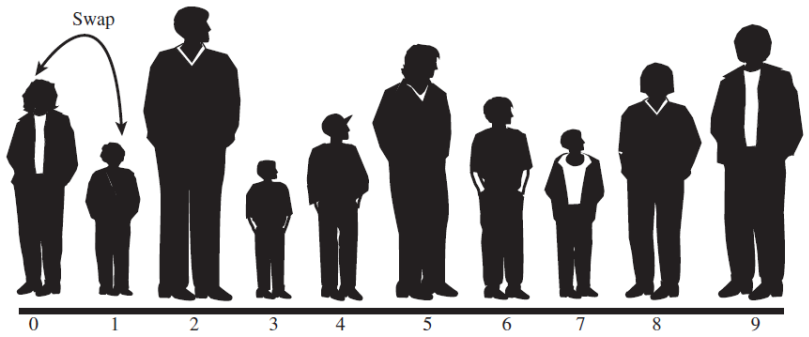
SORTING

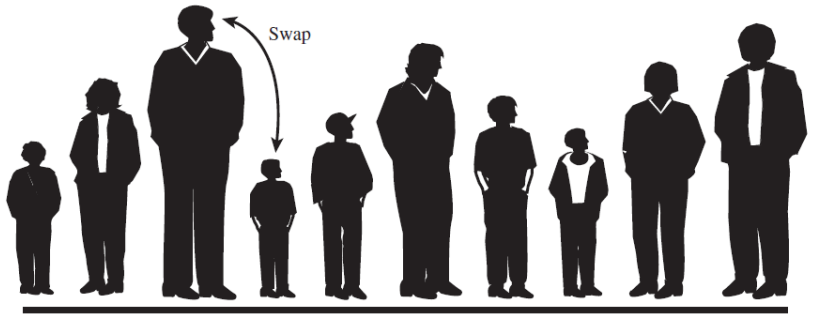
- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.
- Consider an array $A = \{A_1, A_2, \cdots, A_n\}$,
  - The array is called to be in ascending order if $A_1 \leq A_2 \leq \cdots \leq A_n$,
  - Descending, if $A_1 \geq A_2 \geq \cdots \geq A_n$

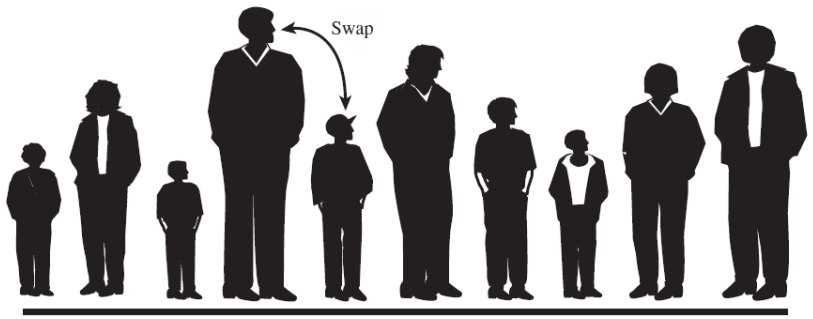
SN	Sorting Algorithms	Description
1	Bubble Sort	It is the simplest sort method which performs sorting by repeatedly moving the largest element to the highest index of the array. It comprises of comparing each element to its adjacent element and replace them accordingly.
2	Bucket Sort	Bucket sort is also known as bin sort. It works by distributing the element into the array also called buckets. In this sorting algorithms, Buckets are sorted individually by using different sorting algorithm.
3	Comb Sort	Comb Sort is the advanced form of Bubble Sort. Bubble Sort compares all the adjacent values while comb sort removes all the turtle values or small values near the end of the list.
4	Counting Sort	It is a sorting technique based on the keys i.e. objects are collected according to keys which are small integers. Counting sort calculates the number of occurrence of objects and stores its key values. New array is formed by adding previous key elements and assigning to objects.
5	Heap Sort	In the heap sort, Min heap or max heap is maintained from the array elements deending upon the choice and the elements are sorted by deleting the root element of the heap.
6	Insertion Sort	As the name suggests, insertion sort inserts each element of the array to its proper place. It is a very simple sort method which is used to arrange the deck of cards while playing bridge.
7	Merge Sort	Merge sort follows divide and conquer approach in which, the list is first divided into the sets of equal elements and then each half of the list is sorted by using merge sort. The sorted list is combined again to form an elementary sorted array.
8	Quick Sort	Quick sort is the most optimized sort algorithms which performs sorting in O(n log n) comparisons. Like Merge sort, quick sort also work by using divide and conquer approach.
9	Radix Sort	In Radix sort, the sorting is done as we do sort the names according to their alphabetical order. It is the lenear sorting algorithm used for Inegers.
10	Selection Sort	Selection sort finds the smallest element in the array and place it on the first place on the list, then it finds the second smallest element in the array and place it on the second place. This process continues until all the elements are moved to their correct ordering. It carries running time O(n2) which is worst than insertion sort.
11	Shell Sort	Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions.

SIMPLE SORTING

- Key idea:
  - Compare two items
  - Swap two items, or copy one item
  - Bubble Sort, Selection Sort, Insertion Sort







Bubble sort: the beginning of the first pass.

Bubble sort: the end of the first pass.

Bubble Sort:

Compare two players.

If the one on the left is taller, swap them.

Move one position right.

When you reach the first sorted player, start over at the left end of the line.

Continue this process until all players are in order.

```
class ArrayBub {
    private long[] a; // ref to array a
    private int nElems; // number of data items
    public ArrayBub(int max) // constructor
    {
        a = new long[max]; // create the array
        nElems = 0; // no items yet
    }
    public void insert(long value) // put element into array
    {
        a[nElems] = value; // insert it
        nElems++; // increment size
    }
    public void display() // displays array contents
    {
        for(int j = 0; j < nElems; j++) // for each element,
            System.out.print(a[j] + " "); // display it
        System.out.println("");
    }
    public void bubbleSort()
    {
        int out, in;
        for(out = nElems - 1; out > 1; out--) // outer loop (backward)
            for(in = 0; in < out; in++) // inner loop (forward)
                if( a[in] > a[in + 1] ) // out of order?
                    swap(in, in + 1); // swap them
    } // end bubbleSort()
    private void swap(int one, int two)
    {
        long temp = a[one];
        a[one] = a[two];
        a[two] = temp;
    }
} // end class ArrayBub
class BubbleSortApp {
    public static void main(String[] args) {
        int maxSize = 100; // array size
        ArrayBub arr; // reference to array
        arr = new ArrayBub(maxSize); // create the array
        arr.insert(77); // insert 10 items
        arr.insert(99);
        arr.insert(44);
        arr.insert(55);
        arr.insert(22);
        arr.insert(88);
        arr.insert(11);
        arr.insert(00);
        arr.insert(66);
        arr.insert(33);
        arr.display(); // display items
        arr.bubbleSort(); // bubble sort them
        arr.display(); // display them again
    } // end main()
} // end class BubbleSortApp
```

```
public void bubbleSort()
   {
   int out, in;

   for(out=nElems-1; out>1; out--)   // outer loop (backward)
      for(in=0; in<out; in++)        // inner loop (forward)
         if( a[in] > a[in+1] )       // out of order?
            swap(in, in+1);          // swap them
   }  // end bubbleSort()
```

```
77 99 44 55 22 88 11 0 66 33
0 11 22 33 44 55 66 77 88 99
```

In general, where N is the number of items in the array, there are N-1 comparisons on the first pass, N-2 on the second, and so on. The formula for the sum of such a series is

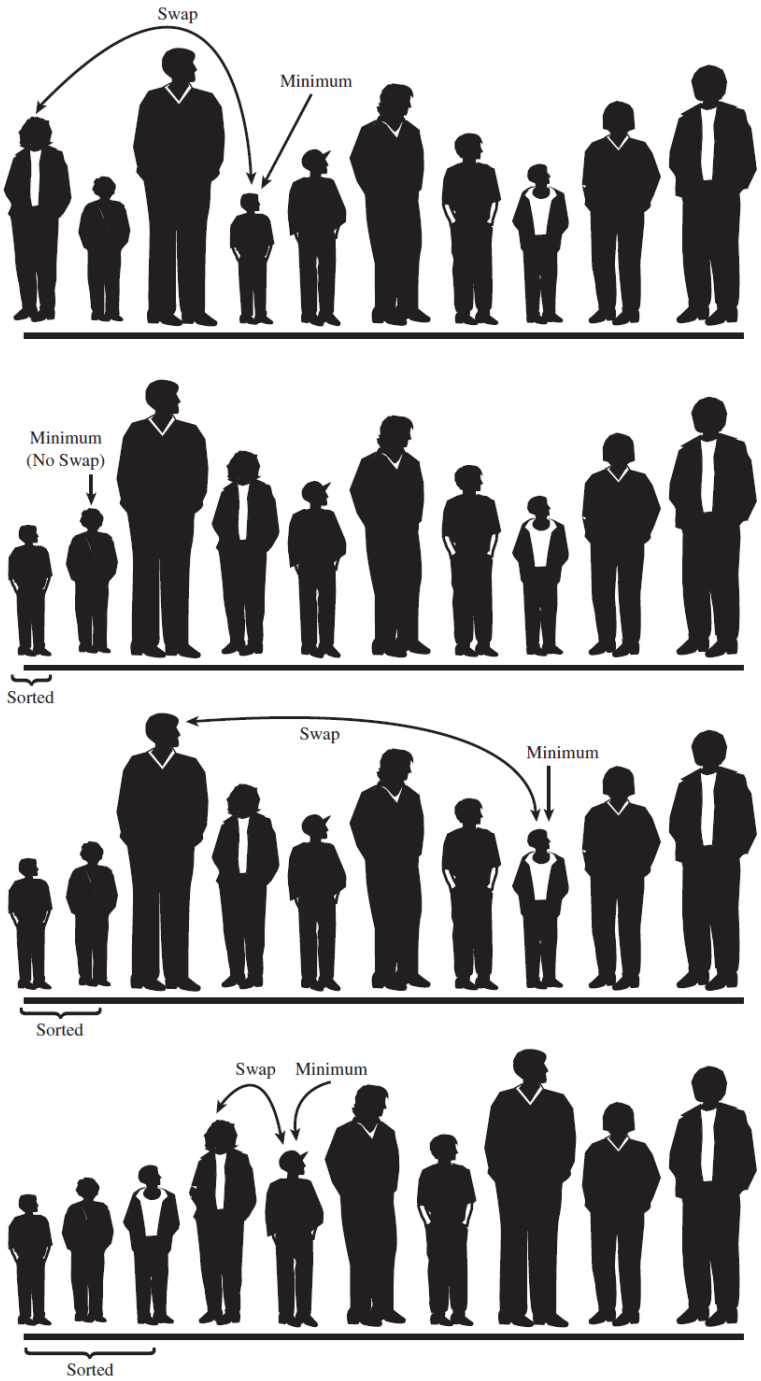
$(N-1) + (N-2) + (N-3) + \ldots + 1 = N*(N-1)/2$

$N*(N-1)/2$ is 45 (10*9/2) when N is 10.

Thus, the algorithm makes about $N^2/2$ comparisons (ignoring the –1, which doesn't make much difference, especially if N is large).

There are fewer swaps than there are comparisons because two bars are swapped only if they need to be. If the data is random, a swap is necessary about half the time, so there will be about $N^2/4$ swaps. (Although in the worst case, with the initial data inversely sorted, a swap is necessary with every comparison.)

Both swaps and comparisons are proportional to N^2. Because constants don't count in Big O notation, we can ignore the 2 and the 4 and say that the bubble sort runs in $O(N^2)$ time.



Selection sort on baseball players.

```
public void selectionSort()
   {
   int out, in, min;

   for(out=0; out<nElems-1; out++)   // outer loop
      {
      min = out;                     // minimum
      for(in=out+1; in<nElems; in++) // inner loop
         if(a[in] < a[min] )         // if min greater,
             min = in;               // we have a new min
      swap(out, min);                // swap them
      }  // end for(out)
   }  // end selectionSort()
```

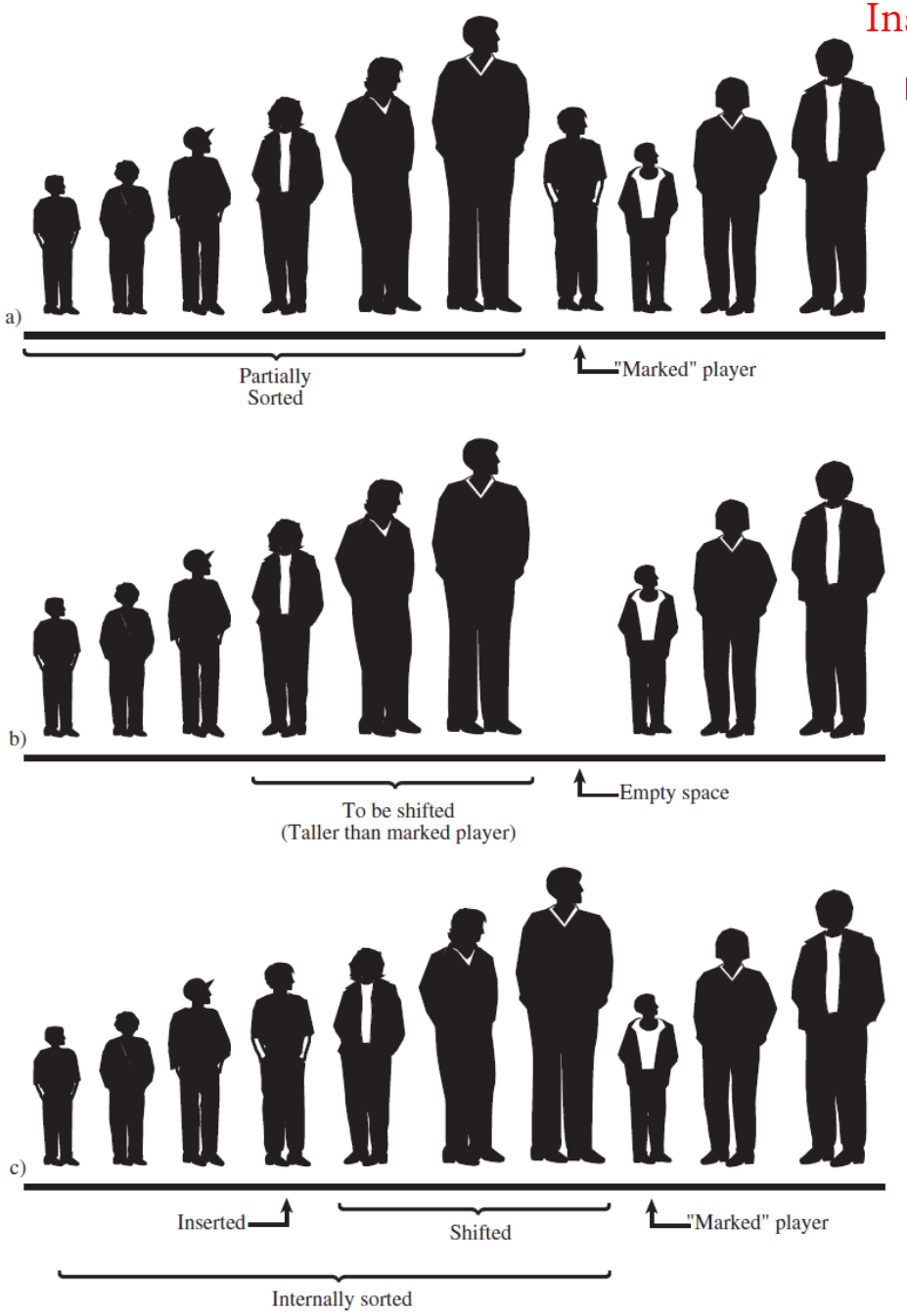
Selection Sort:

Making a pass through all players and picking the shortest one. This shortest player is then swapped with the player on the left end of the line, at position 0. Now the leftmost player is sorted and won't need to be moved again.

The next time you pass down the players, you start at position 1, and, finding the minimum, swap with position 1. This process continues until all players are sorted.

The selection sort performs the same number of comparisons as the bubble sort: N*(N-1)/2. For 10 data items, this is 45 comparisons. However, 10 items require fewer than 10 swaps. With 100 items, 4,950 comparisons are required, but fewer than 100 swaps. For large values of N, the comparison times will dominate, so we would have to say that the selection sort runs in $O(N^2)$ time, just as the bubble sort did. However, it is unquestionably faster because there are so few swaps. For smaller values of N, the selection sort may in fact be considerably faster, especially if the swap times are much larger than the comparison times.

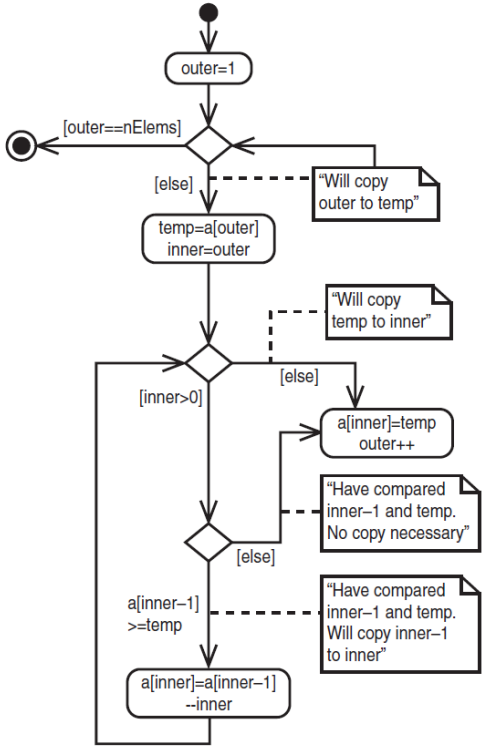
Insertion Sort



The insertion sort on baseball players.

```
public void insertionSort()
   {
   int in, out;

   for(out=1; out<nElems; out++)     // out is dividing line
      {
      long temp = a[out];            // remove marked item
      in = out;                      // start shifts at out
      while(in>0 && a[in-1] >= temp) // until one is smaller,
         {
         a[in] = a[in-1];            // shift item to right
         --in;                       // go left one position
         }
      a[in] = temp;                  // insert marked item
      }  // end for
```



Activity diagram for `insertSort()`.

In most cases the insertion sort is the best of the elementary sorts described in this chapter. It still executes in $O(N^2)$ time, but it's about twice as fast as the bubble sort and somewhat faster than the selection sort in normal situations.

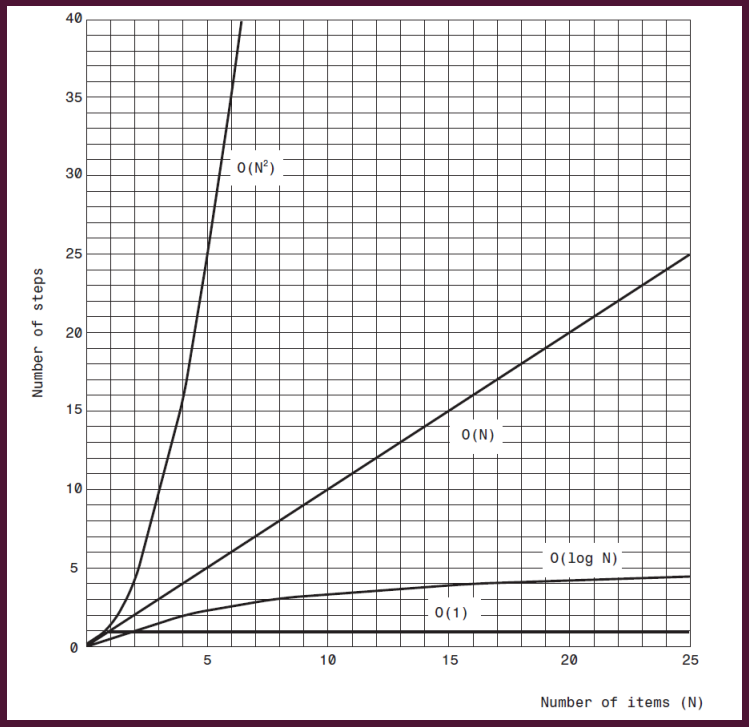
SUMMARY OF SIMPLE SORTING

- The sorting algorithms all assume an array as a data storage structure.
- Sorting involves comparing data items in the array and moving them until sorted.
- All execute in $O(N^2)$ time. Nevertheless, some can be substantially faster than others.
- An invariant is a condition that remains unchanged while an algorithm runs.
- The bubble sort is the least efficient, but the simplest, sort.
- The insertion sort is the most commonly used of the $O(N^2)$ sorts.
- A sort is stable if the order of elements with the same key is retained.
- None of the sorts require more than a single temporary variable, in addition to the original array.

ADVANCED SORTING

- Merge Sort, Shellsort, and Quick Sort
  - Operate much faster than simple sorting

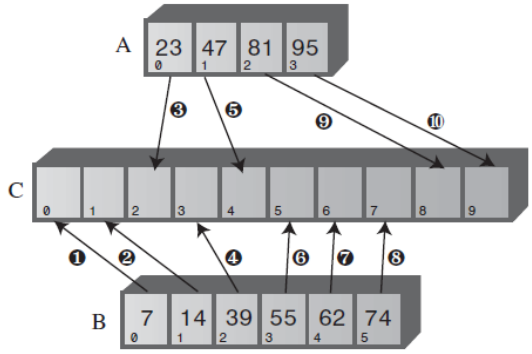
MERGESORT



- A much more efficient sorting technique than simple sorting methods, at least in terms of speed.
  - Bubble, Insertion, and Selection sorts take $O(N^2)$ time.
  - Mergesort is $O(N * \log N)$.
  - $N = 10{,}000$: $N^2 = 100{,}000{,}000$, $N * \log N = 40{,}000$.
  - If sorting this N items required 40 seconds with Mergesort, it would take almost 28 hours for Insertion sort.
  - It's conceptually easier than Quicksort and Shell sort.
- Mergesort requires an additional array, equal in size to the one being sorted.
  - With limited memory, Mergesort won't work.
  - If you have enough space, it's a good choice.

MERGING TWO SORTED ARRAYS

- The heart of Mergesort is the merging of two already-sorted arrays.
- Merging two sorted arrays A and B creates a third array C, that contains all the elements of A and B, also arranged in sorted order.



a) Before Merge

b) After Merge

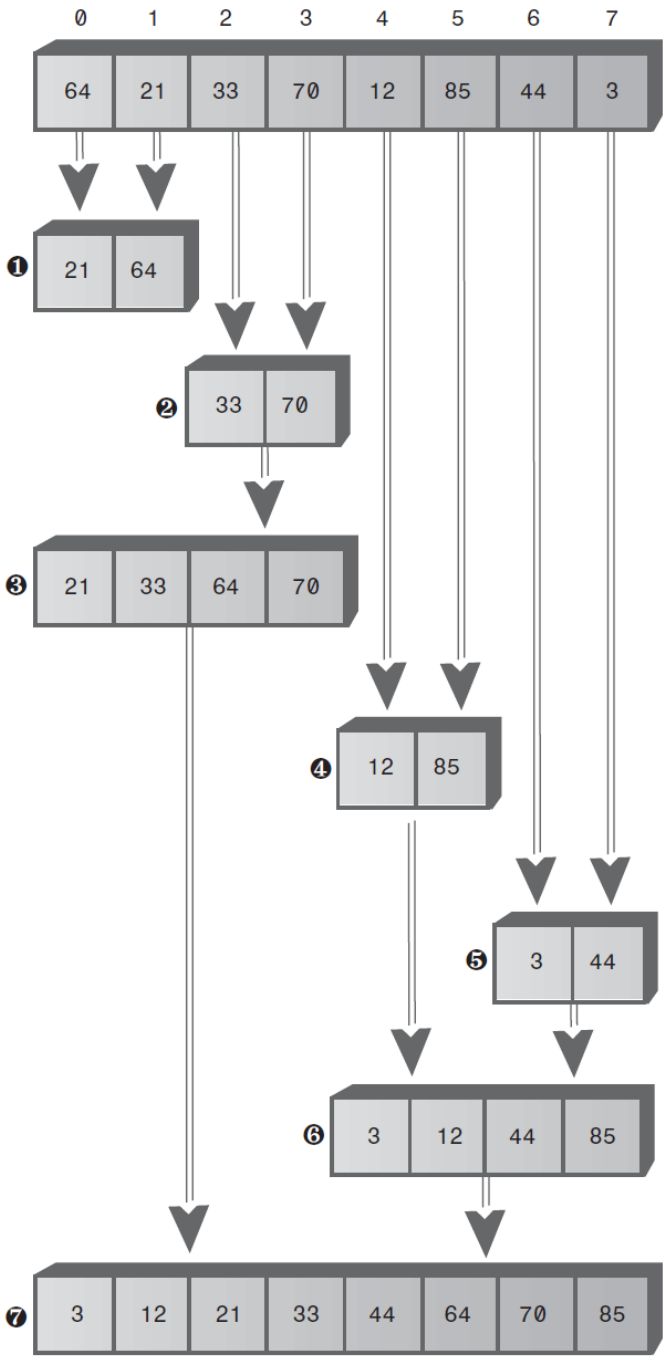
Step	Comparison (If Any)	Copy
1	Compare 23 and 7	Copy 7 from B to C
2	Compare 23 and 14	Copy 14 from B to C
3	Compare 23 and 39	Copy 23 from A to C
4	Compare 39 and 47	Copy 39 from B to C
5	Compare 55 and 47	Copy 47 from A to C
6	Compare 55 and 81	Copy 55 from B to C
7	Compare 62 and 81	Copy 62 from B to C
8	Compare 74 and 81	Copy 74 from B to C
9		Copy 81 from A to C
10		Copy 95 from A to C

```java
// merge.java
// demonstrates merging two arrays into a third
// to run this program: C>java MergeApp
////////////////////////////////////////////////////////////////
class MergeApp
{
    public static void main(String[] args)
    {
        int[] arrayA = {23, 47, 81, 95};
        int[] arrayB = {7, 14, 39, 55, 62, 74};
        int[] arrayC = new int[10];
        merge(arrayA, 4, arrayB, 6, arrayC);
        display(arrayC, 10);
    } // end main()
    //--------------------------------------------------------------
    // merge A and B into C
    public static void merge( int[] arrayA, int sizeA,
                              int[] arrayB, int sizeB,
                              int[] arrayC )
    {
        int aDex = 0, bDex = 0, cDex = 0;
        while(aDex < sizeA && bDex < sizeB) // neither array empty
            if( arrayA[aDex] < arrayB[bDex] )
                arrayC[cDex++] = arrayA[aDex++];
            else
                arrayC[cDex++] = arrayB[bDex++];
        while(aDex < sizeA) // arrayB is empty,
            arrayC[cDex++] = arrayA[aDex++]; // but arrayA isn't
        while(bDex < sizeB) // arrayA is empty,
            arrayC[cDex++] = arrayB[bDex++]; // but arrayB isn't
    } // end merge()
    //--------------------------------------------------------------
    // display array
    public static void display(int[] theArray, int size)
    {
        for(int j = 0; j < size; j++)
            System.out.print(theArray[j] + " ");
        System.out.println("");
    }
    //--------------------------------------------------------------
} // end class MergeApp
```

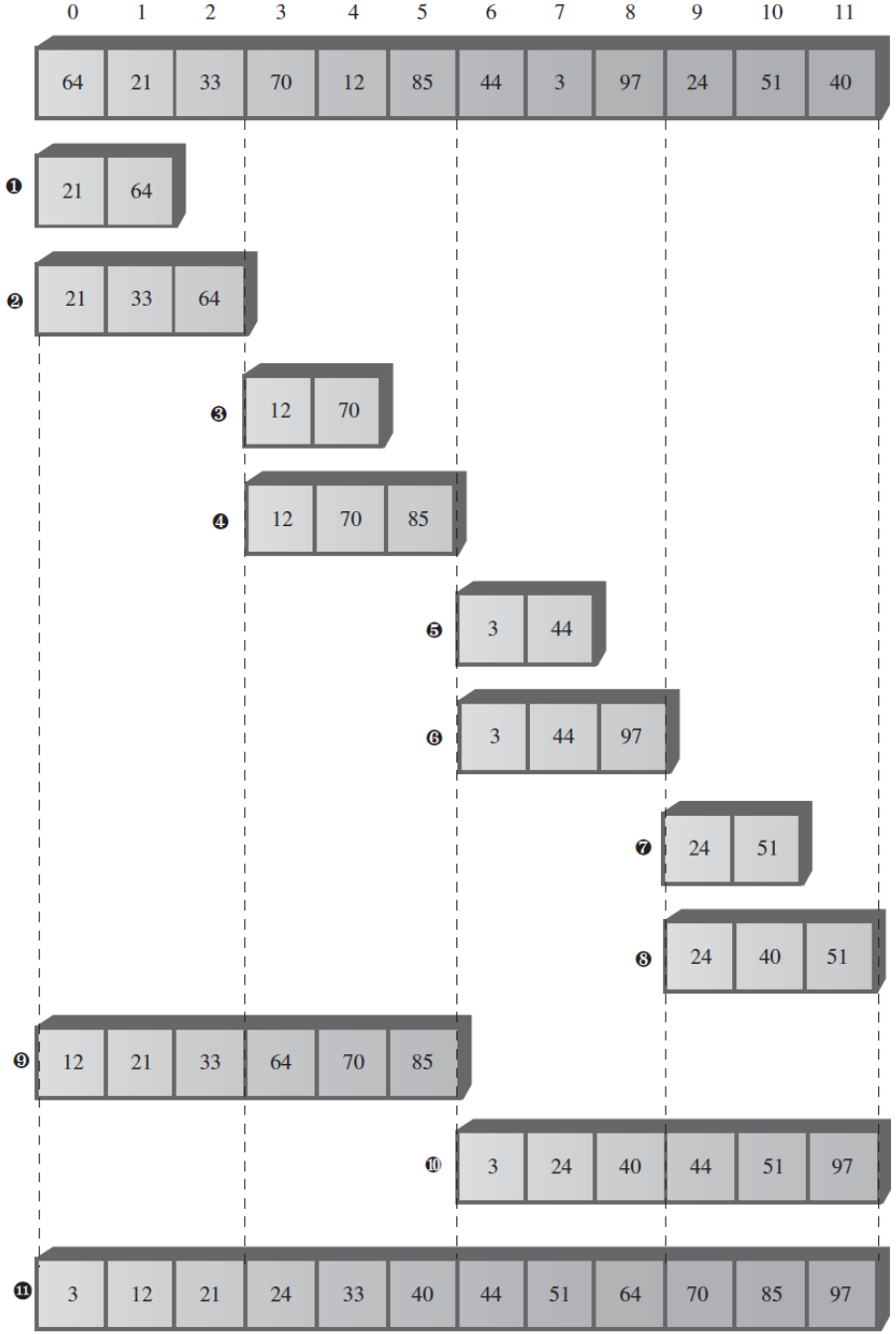
SORTING BY MERGING

- The idea in Mergesort:
  - divide an array in half,
  - sort each half,
  - and then use the merge() method to merge the two halves into a single sorted array.

- How to sort each half?
  - divide the half into two quarters,
  - sort each of the quarters,
  - and merge them to make a sorted half.



Merging larger and larger arrays.



Array size not a power of 2.

```java
// mergeSort.java
// demonstrates recursive merge sort
// to run this program: C>java MergeSortApp
////////////////////////////////////////////////////////////////
class DArray
{
    private long[] theArray; // ref to array theArray
    private int nElems; // number of data items
    //--------------------------------------------------------------
    public DArray(int max) // constructor
    {
        theArray = new long[max]; // create array
        nElems = 0;
    }
    //--------------------------------------------------------------
    public void insert(long value) // put element into array
    {
        theArray[nElems] = value; // insert it
        nElems++; // increment size
    }
    //--------------------------------------------------------------
    public void display() // displays array contents
    {
        for(int j = 0; j < nElems; j++) // for each element,
            System.out.print(theArray[j] + " "); // display it
        System.out.println("");
    }
    //--------------------------------------------------------------
    public void mergeSort() // called by main()
    {
        // provides workspace
        long[] workSpace = new long[nElems];
        recMergeSort(workSpace, 0, nElems - 1);
    }
    //--------------------------------------------------------------
    private void recMergeSort(long[] workSpace, int lowerBound,
                              int upperBound)
    {
        if(lowerBound == upperBound) // if range is 1,
            return; // no use sorting
        else
        {
            // find midpoint
            int mid = (lowerBound + upperBound) / 2;
            // sort low half
            recMergeSort(workSpace, lowerBound, mid);
            // sort high half
            recMergeSort(workSpace, mid + 1, upperBound);
            // merge them
            merge(workSpace, lowerBound, mid + 1, upperBound);
        } // end else
    } // end recMergeSort()
    //--------------------------------------------------------------
    private void merge(long[] workSpace, int lowPtr,
                       int highPtr, int upperBound)
    {
        int j = 0; // workspace index
        int lowerBound = lowPtr;
        int mid = highPtr - 1;
        int n = upperBound - lowerBound + 1; // # of items
        while(lowPtr <= mid && highPtr <= upperBound)
            if( theArray[lowPtr] < theArray[highPtr] )
                workSpace[j++] = theArray[lowPtr++];
            else
                workSpace[j++] = theArray[highPtr++];
        while(lowPtr <= mid)
            workSpace[j++] = theArray[lowPtr++];
        while(highPtr <= upperBound)
            workSpace[j++] = theArray[highPtr++];
        for(j = 0; j < n; j++)
            theArray[lowerBound + j] = workSpace[j];
    } // end merge()
    //--------------------------------------------------------------
} // end class DArray
////////////////////////////////////////////////////////////////
class MergeSortApp
{
    public static void main(String[] args)
    {
        int maxSize = 100; // array size
        DArray arr; // reference to array
        arr = new DArray(maxSize); // create the array
        arr.insert(64); // insert items
        arr.insert(21);
        arr.insert(33);
        arr.insert(70);
        arr.insert(12);
        arr.insert(85);
        arr.insert(44);
        arr.insert(3);
        arr.insert(99);
        arr.insert(0);
        arr.insert(108);
        arr.insert(36);
        arr.display(); // display items
        arr.mergeSort(); // merge sort the array
        arr.display(); // display items again
    } // end main()
} // end class MergeSortApp
```

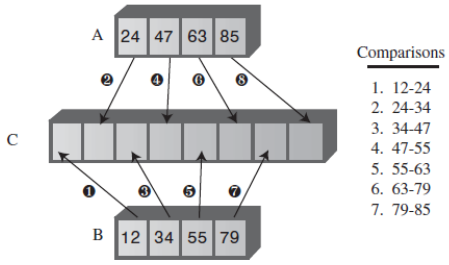
EFFICIENCY OF MERGESORT

- Mergesort runs in $O(N * \log N)$ time. Why?
  - Let's figure out the number of times a data item must be copied and the number times it must be compared with another data item.
  - Assume that copying/comparing are the most expensive operations; that the recursive calls and returns don't add much overhead.

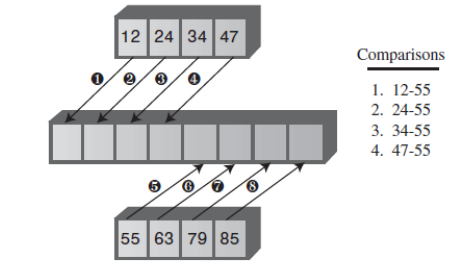
- Example: To sort 8 items requires 3 levels, each of which involves 8 copies. A level means all copies into the same size subarray.
  - In the 1st level, there are four 2-element subarrays;
  - In the 2nd level, there are two 4-element subarrays;
  - In the 3rd level, there is one 8-element subarray.
  - Each level has 8 elements, and there are 3*8 or 24 copies.

Number of Operations When N Is a Power of 2

N	$\log_2 N$	Number of Copies Into Workspace ($N*\log_2 N$)	Total Copies	Comparisons Max (Min)
2	1	2	4	1 (1)
4	2	8	16	5 (4)
8	3	24	48	17 (12)
16	4	64	128	49 (32)
32	5	160	320	129 (80)
64	6	384	768	321 (192)
128	7	896	1792	769 (448)



a) Worst-case Scenario



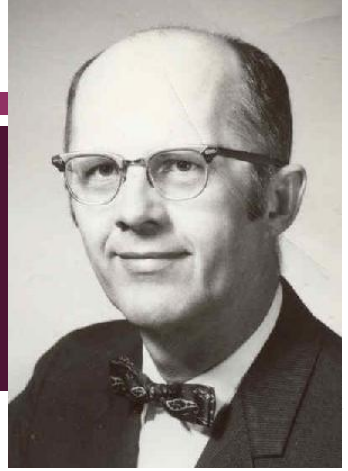
b) Best-case Scenario

Step Number	1	2	3	4	5	6	7	Totals
Number of items being merged (N)	2	2	4	2	2	4	8	24
Maximum comparisons (N-1)	1	1	3	1	1	3	7	17
Minimum comparisons (N/2)	1	1	2	1	1	2	4	12

SUMMARY OF MERGESORT

- Merging two sorted arrays means to create a third array that contains all the elements from both arrays in sorted order.
- In mergesort, 1-element subarrays of a larger array are merged into 2-element subarrays, 2-element subarrays are merged into 4-element subarrays, and so on until the entire array is sorted.
- Mergesort requires $O(N * \log N)$ time.
- Mergesort requires a workspace equal in size to the original array.

SHELLSORT



SCIENTIFIC AND BUSINESS APPLICATIONS

A High-Speed Sorting Procedure

D. L. Shell, *General Electric Company, Cincinnati, Ohio*

There are a number of methods that have been used for sorting purposes in various machine programs from time to time. Most of these methods are reviewed by Harold Seward [1] in his thesis. One tacit assumption runs through his entire discussion of internal sorting procedures, namely, that the internal memory is relatively small. In other words, the number of items to be sorted is so large that they cannot possibly all fit into the memory at one time.

The methods of internal sorting which he discusses are sorting by:

1) Finding the smallest.
2) Interchanging pairs.
3) Sifting.
4) Partial sort.
5) Merging pairs.
6) Floating decimal sort.

The first four methods all require a time proportional to n^2, where n is the number of items being sorted. The time for the fifth method is proportional to $n(\ln n)$. The time for the sixth method is proportional to $n(\ln r)$, where r is the largest number to be used in a key.

As pointed out in Seward's paper, one would normally choose either method five or six for a rapid internal sort, especially if n is to be very large. The chief drawback of these two methods, however, is the fact that they require twice as much storage as the other four methods.

The advent of very large high-speed random access memories changes the picture relative to sorting somewhat. It is now possible to have a very large number of items to be sorted in memory all at one time. It is highly desirable, therefore, to have a method with the speed characteristics of the merging by pairs and the space characteristics of sifting. If such a method were available it would be possible to sort twice as many items at one time in the machine and still do it at a reasonably high speed.

Such a method is outlined in this paper. The idea is, in part, to combine some of the properties of merging with some of the properties of sifting. The method is most easily described by reference to the block diagram, figure 1. Suppose we are given a sequence of elements f_i to be

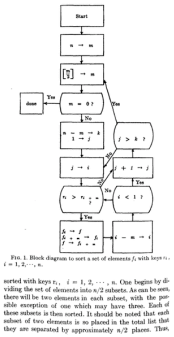


FIG. 1. Block diagram to sort a set of elements f_i with keys r_i, $i = 1, 2, \cdots, n$.

sorted with keys r_i, $i = 1, 2, \cdots, n$. One begins by dividing the set of elements into $n/2$ subsets. As can be seen, there will be two elements in each subset, with the possible exception of one which may have three. Each of these subsets is then sorted. It should be noted that each subset of two elements is so placed in the total list that they are separated by approximately $n/2$ places. Then

- Shellsort: named for Donald L. Shell, the computer scientist who discovered it in 1959.
  - It's based on insertion sort, with a new feature that dramatically improves performance.
  - It's good for medium-sized arrays, depending on implementation.
  - It's not quite as fast as quicksort, so it's not optimum for very large files.
  - It's much faster than the $O(N^2)$ sorts.
  - It's very easy to implement: short and simple.

- The worst-case performance is not significantly worse than the average performance.
- Start with Shellsort and change to others only if it proves too slow in practice.

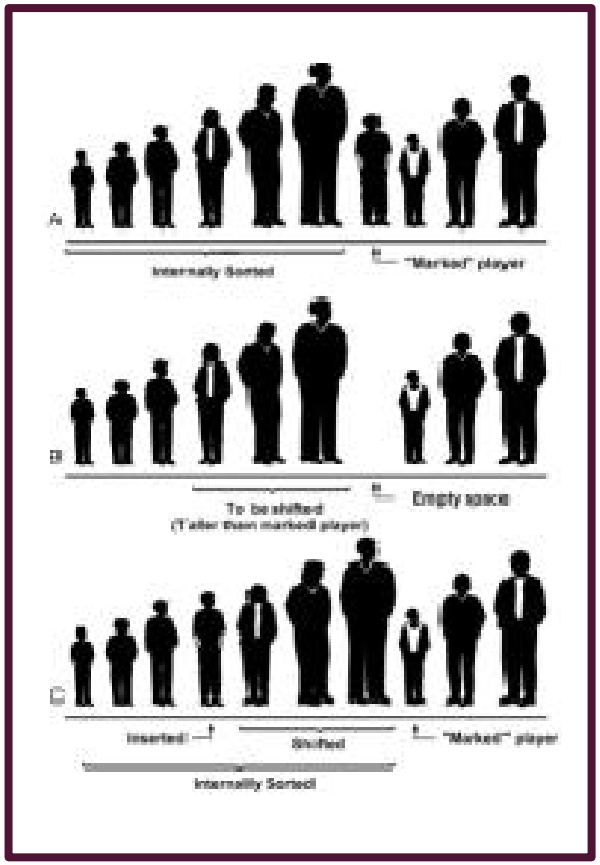
PROBLEM WITH INSERTION SORT

- Recall the insertion sort:
  - The items to the left of a marker are internally sorted and items to the right are not.
  - The algorithm removes the item at the marker and stores it in a temporary variable.
  - Beginning with the item to the left of the newly vacated cell, it shifts the sorted items right one cell at a time, until the item in the temporary variable can be reinserted in sorted order.

- Problem with the insertion sort.
  - Suppose a small item is on the far right. To move this small item to its proper place on the left, all intervening items must be shifted one space right. This step takes close to N copies, just for one item.
  - Not all items must be moved N spaces, but the average item must be moved $N/2$ spaces, which takes N times $N/2$ shifts for totally $N^2/2$ copies. Thus, the performance of insertion sort is $O(N^2)$.

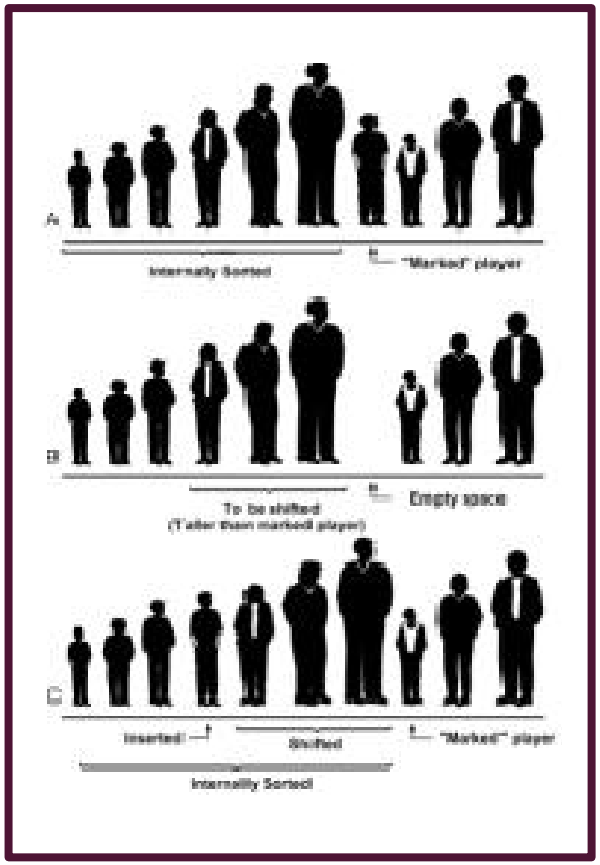
- Idea: moving a smaller item many spaces to the left without shifting all intermediate items individually.

RECALL INSERTION SORT....



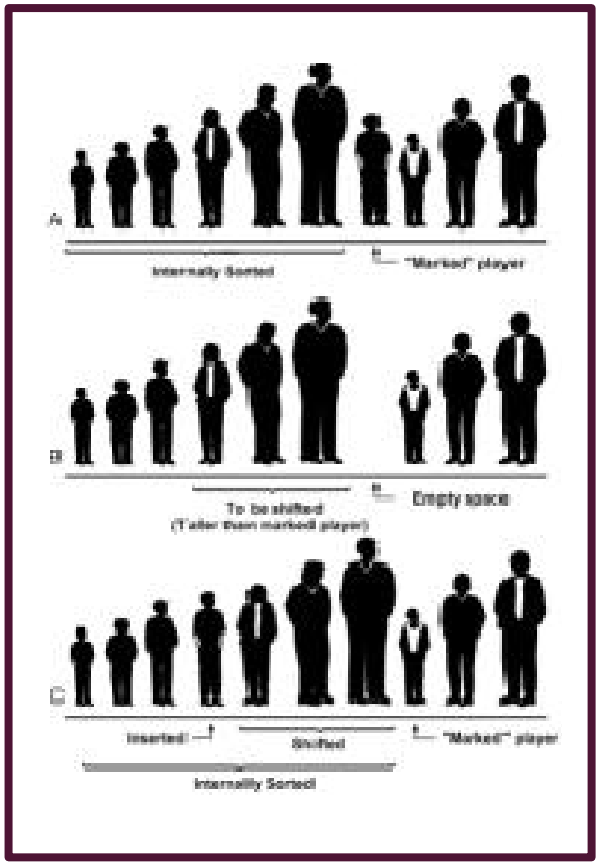
- A subarray to the left is 'partially sorted'
  - Start with the first element
- The player immediately to the right is 'marked'.
- The 'marked' player is inserted into the correct place in the partially sorted array
  - Remove first
  - Marked player 'walks' to the left
  - Shift appropriate elements until we hit a smaller one

THE PROBLEM



- If a small item is very far to the right
  - Like in this case ->
- You must shift many intervening large items one space to the right
  - Almost N copies
  - Average case N/2
  - N items, $N^2/2$ copies
- Better if:
  - Move a small item many spaces, without shifting

REMEMBER



- What made insertion sort the best of the basic sorts?
  - If the array is almost sorted, O(N)
- Shellsort has two steps:
  - "Almost sort" the array
  - Run insertion sort

THE "ALMOST SORT" STEP

- Say we have a 10-element array:
  - 60 30 80 90 0 20 70 10 40 50
- Sort indices 0, 4, and 8:
  - 0 30 80 90 40 20 70 10 60 50
- Sort indices 1, 5, and 9:
  - 0 20 80 90 40 30 70 10 60 50
- Sort indices 2, 6:
  - 0 20 70 90 40 30 80 10 60 50
- Sort indices 3, 7
  - 0 20 70 10 40 30 80 90 60 50

THE "ALMOST SORT" STEP

- This is called a "4-sort":
  - 60 30 80 90 0 20 70 10 40 50
  - 0 30 80 90 40 20 70 10 60 50
  - 0 20 80 90 40 30 70 10 60 50
  - 0 20 70 90 40 30 80 10 60 50
  - 0 20 70 10 40 30 80 90 60 50

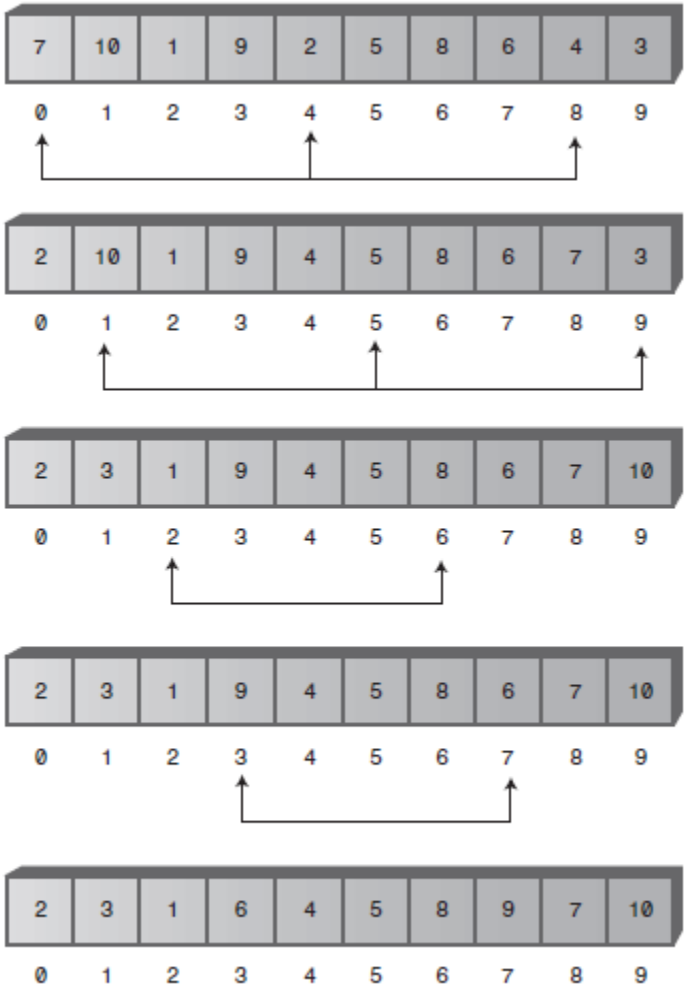
- Once we've done this, the array is almost sorted, and we can run insertion sort on the whole thing
  - Should be about O(N) time

N-SORT

- Shellsort achieves large shifts by insertion-sorting widely spaced elements. Then it sorts less widely spaced elements.
- The spacing between elements for these sorts is called the increment and is represented by a letter h.

- Example: sorting a 10-element array with an increment of 4.
- Elements 0, 4, and 8 are sorted.
- Then the algorithm shifts over one cell and sorts 1, 5, and 9.
- This process continues until all elements are 4-sorted.

- After 4-sort, the array becomes four subarrays: (0,4,8), (1,5,9), (2,6), and (3,7), each completely sorted.



A complete 4-sort.

Insertion sort is efficient when operating on an array that's almost sorted. If it needs to move items only one or two cells to sort, it can operate in almost $O(N)$ time. Thus, after the array has been 4-sorted, we can 1-sort it using the ordinary insertion sort. Combining 4-sort and 1-sort is much faster than applying the ordinary insertion sort without 4-sort.

DIMINISHING GAPS

- In Shellsort, the interval is repeatedly reduced until it becomes 1.
  - An array of 1,000 items might be 364-sorted, then 121-sorted, then 40-sorted, then 13-sorted, then 4-sorted, and finally 1-sorted.
  - The sequence of numbers is called the interval sequence or gap sequence.

- Starting from 1, it's generated by
  - $h = 3 \times h + 1$

Knuth's Interval Sequence

h	3*h + 1	(h–1) / 3
1	4	
4	13	1
13	40	4
40	121	13
121	364	40
364	1093	121
1093	3280	364

```java
// shellSort.java
// demonstrates shell sort
// to run this program: C>java ShellSortApp
//--------------------------------------------------------------
class ArraySh {
    private long[] theArray; // ref to array theArray
    private int nElems; // number of data items
    //--------------------------------------------------------------
    public ArraySh(int max) { // constructor
        theArray = new long[max]; // create the array
        nElems = 0; // no items yet
    }
    //--------------------------------------------------------------
    public void insert(long value) { // put element into array
        theArray[nElems] = value; // insert it
        nElems++; // increment size
    }
    //--------------------------------------------------------------
    public void display() { // displays array contents
        System.out.print("A = ");
        for(int j = 0; j < nElems; j++) // for each element,
            System.out.print(theArray[j] + " "); // display it
        System.out.println("");
    }
    //--------------------------------------------------------------
    public void shellSort() {
        int inner, outer;
        long temp;
        int h = 1; // find initial value of h
        while(h <= nElems / 3)
            h = h * 3 + 1; // (1, 4, 13, 40, 121, ...)
        while(h > 0) { // decreasing h, until h=1
            // h-sort the file
            for(outer = h; outer < nElems; outer++) {
                temp = theArray[outer];
                inner = outer;
                // one subpass (eg 0, 4, 8)
                while(inner > h - 1 && theArray[inner - h] >= temp) {
                    theArray[inner] = theArray[inner - h];
                    inner -= h;
                }
                theArray[inner] = temp;
            } // end for
            h = (h - 1) / 3; // decrease h
        } // end while(h>0)
    } // end shellSort()
    //--------------------------------------------------------------
} // end class ArraySh
```

```java
class ShellSortApp {
    public static void main(String[] args) {
        int maxSize = 10; // array size
        ArraySh arr;
        arr = new ArraySh(maxSize); // create the array
        for(int j = 0; j < maxSize; j++) {
            // fill array with random numbers
            long n = (int)(java.lang.Math.random() * 99);
            arr.insert(n);
        }
        arr.display(); // display unsorted array
        arr.shellSort(); // shell sort the array
        arr.display(); // display sorted array
    } // end main()
} // end class ShellSortApp
```

```
A=20 89 6 42 55 59 41 69 75 66
A=6 20 41 42 55 59 66 69 75 89
```

EFFICIENCY OF SHELLSORT

- No theoretical results on its efficiency, except in special cases.
- Various estimates based on experiments, from $O(N^{\frac{3}{2}})$ down to $O(N^{\frac{7}{6}})$.

Estimates of Shellsort Running Time

O() Value	Type of Sort	10 Items	100 Items	1,000 Items	10,000 Items
N^2	Insertion, etc.	100	10,000	1,000,000	100,000,000
$N^{3/2}$	Shellsort	32	1,000	32,000	1,000,000
$N*(\log N)^2$	Shellsort	10	400	9,000	160,000
$N^{5/4}$	Shellsort	18	316	5,600	100,000
$N^{7/6}$	Shellsort	14	215	3,200	46,000
N*logN	Quicksort, etc.	10	200	3,000	40,000

SUMMARY OF SHELLSORT

- Shellsort applies insertion sort to widely spaced elements, then less widely spaced...
- The expression n-sorting means sorting every n-th element.
- A sequence of numbers is used to determine the sorting intervals.
- A widely used interval sequence is generated by $h = 3 \times h + 1$, where initially $h = 1$.
- With 1,000 items, it could be 364, 121, 40, 13, 4, and finally 1-sorted.
- The Shellsort is hard to analyze, but runs in approximately $O(N \times (\log N)^2)$ time.
  - much faster than the $O(N^2)$ algorithms like insertion sort,
  - but slower than the $O(N \times \log N)$ algorithms like quicksort.

QUICKSORT

- A popular sorting algorithm discovered by C.A.R. Hoare in 1962
  - In many situations, it's the fastest, operating in $O(N * \log N)$ time (for in-memory sorting)
- Basic scheme
  - The algorithm operates by partitioning an array into two subarrays.
  - The algorithm then calls itself recursively to quicksort each of these subarrays.
- Some embellishments we can make
  - selection of the pivot
  - sorting of small partitions

PARTITIONING

- Idea: Divide data into two groups, such that:
  - All items with a key value higher than a specified amount (the pivot) are in one group
  - All items with a lower key value are in another
- Applications:
  - Divide employees who live within 15 miles of the office with those who live farther away
  - Divide households by income for taxation purposes
  - Divide computers by processor speed
- Let's see an example with an array

PARTITIONING

- Say I have 12 values:
  - **175 192 95 45 115 105 20 60 185 5 90 180**
- I pick a pivot=104, and partition (NOT sorting yet):
  - **95 45 20 60 5 90** | 175 192 115 105 185 180
  - Note: In the future the pivot will be an actual element
  - Also: Partitioning need not maintain order of elements and usually won't
    - Although I did in this example
- The **partition** is the leftmost item in the right array:
  - **95 45 20 60 5 90** | **175** 192 115 105 185 180
- Which we return to designate where the division is located.

EFFICIENCY: PARTITIONING

- *O(n)* time
  - left starts at 0 and moves one-by-one to the right
  - right starts at n-1 and moves one-by-one to the left
  - When left and right cross, we stop.
    - So we'll hit each element just once

- Number of comparisons is *n+1*
- Number of swaps is worst case *n/2*
  - Worst case, we swap every single time
  - Each swap involves two elements
  - Usually, it will be less than this
    - Since in the random case, some elements will be on the correct side of the pivot

MODIFIED PARITITIONING

- In preparation for Quicksort:
  - Choose our pivot value to be the rightmost element
  - Partition the array around the pivot
  - Ensure the pivot is at the location of the partition
    - Meaning, the pivot should be the leftmost element of the right subarray

- Example:
  - Unpartitioned: **42 89 63 12 94 27 78 10 50 36**
  - Partitioned around Pivot: **3 27 12 36** 63 94 89 78 42 50

- What does this imply about the pivot element after the partition?

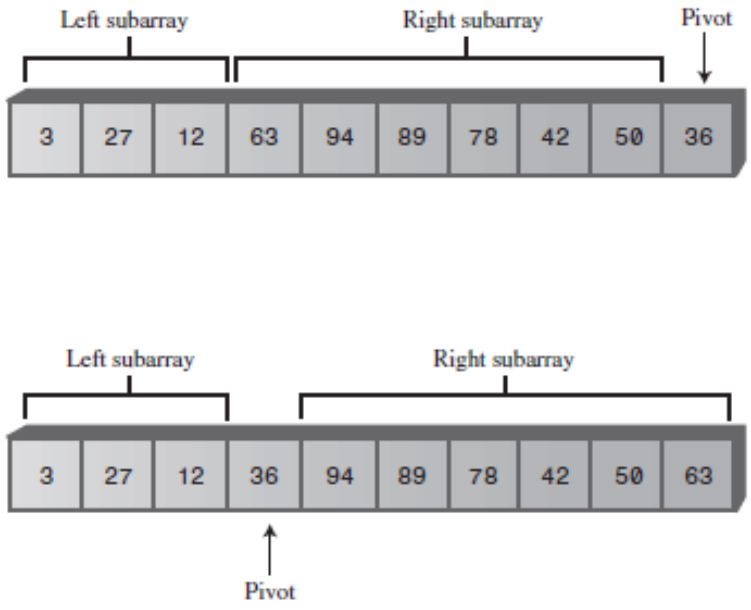
PLACING THE PIVOT

- Goal: Pivot must be in the leftmost position in the right subarray
  - 3 27 12 36 63 94 89 78 42 50
- Our algorithm does not do this currently.
- It currently will not touch the pivot
  - left increments till it finds an element < pivot
  - right decrements till it finds an element > pivot
  - So the pivot itself won't be touched, and will stay on the right:
  - 3 27 12 63 94 89 78 42 50 36

OPTIONS

- We have this:
  - 3 27 12 63 94 89 78 42 50 36
- Our goal is the position of 36:
  - 3 27 12 36 63 94 89 78 42 50
- We could either:
  - Shift every element in the right subarray up (inefficient)
  - Just swap the leftmost with the pivot! Better ☺
  - We can do this because the right subarray is not in any particular order
  - 3 27 12 36 94 89 78 42 50 63

SWAPPING THE PIVOT



Swapping the pivot.

- Just takes one more line to our Java method
- Basically, a single call to swap()
  - Swaps A[end-1] (the pivot) with A[left] (the partition index)

PARTITIONING

- The partition process
  - Start with two pointers: *leftPtr* initialized to one position to the left of the first cell; *rightPtr* to one position to the right of the last cell.
  - *leftPtr* moves to the right; *rightPtr* moves to the left.
- Stopping and Swapping
  - When *leftPtr* encounters an item smaller than the pivot, it keeps going; when it finds a larger item, it stops.
  - When *rightPtr* encounters an item larger than the pivot, it keeps going; when it finds a smaller item, it stops.
- When the two pointers eventually meet, the process is complete.

```
public int partitionIt(int left, int right, long pivot)
   {
   int leftPtr = left-1;           // left    (after ++)
   int rightPtr = right;           // right-1 (after --)
   while(true)
      {                            // find bigger item
      while( theArray[++leftPtr] < pivot )
         ;  // (nop)
                                   // find smaller item
      while(rightPtr > 0 && theArray[--rightPtr] > pivot)
         ;  // (nop)

      if(leftPtr >= rightPtr)      // if pointers cross,
         break;                    //    partition done
      else                         // not crossed, so
         swap(leftPtr, rightPtr);  //    swap elements
      }  // end while(true)
   swap(leftPtr, right);           // restore pivot
   return leftPtr;                 // return pivot location
   }  // end partitionIt()
```

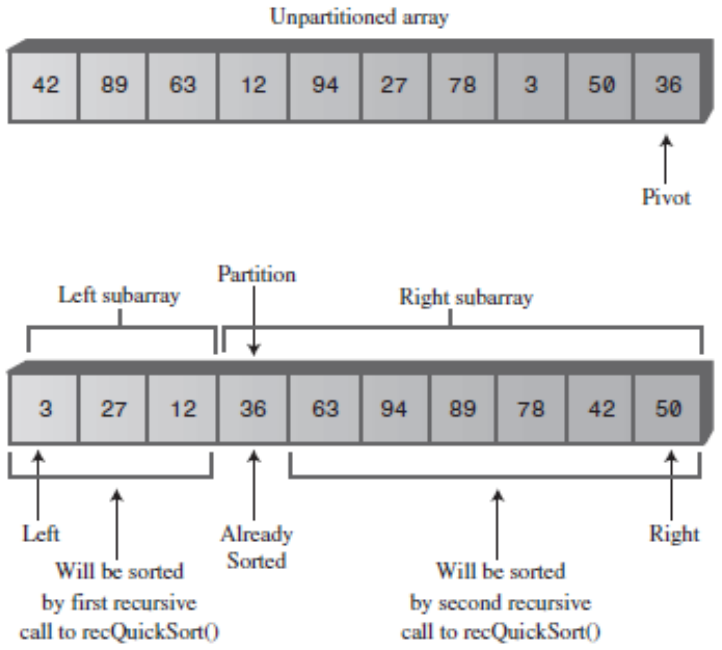
QUICKSORT

- The most popular sorting algorithm
- For most situations, it runs in $O(N * \log N)$
  - Remember partitioning. It's the key step. And it's $O(n)$.

- The basic algorithm (recursive):
  - Partition the array into left (smaller keys) and right (larger keys) groups
  - Call ourselves and sort the left group
  - Call ourselves and sort the right group

  - Base case: We partition just one element, which is just the element itself

```
public void recQuickSort(int left, int right)
      {
      if(right-left <= 0)        // if size <= 1,
          return;                //    already sorted
      else                       // size is 2 or larger
         {
         long pivot = theArray[right];      // rightmost item
                                            // partition range
         int partition = partitionIt(left, right, pivot);
         recQuickSort(left, partition-1);   // sort left side
         recQuickSort(partition+1, right);  // sort right side
         }
      }  // end recQuickSort()
```



Recursive calls sort subarrays.

QUICKSORT

- Partition array/subarray into left (smaller keys) and right (larger keys) groups.
- Call ourselves to sort the left group.
- Call ourselves to sort the right group.

SHALL WE TRY IT ON AN ARRAY?

- 10 70 50 30 60 90 0 40 80 20
- Let's go step-by-step on the board

BEST CASE…

- We partition the array each time into two equal subarrays
- Say we start with array of size $n = 2^i$
- We recurse until the base case, 1 element

- Draw the tree
  - First call -> Partition n elements, n operations
  - Second calls -> Each partition n/2 elements, 2(n/2)=n operations
  - Third calls -> Each partition n/4, 4(n/4) = n operations
  - …
  - (i+1)th calls -> Each partition $n/2^i = 1$, $2^i(1) = n(1) = n$ ops

- Total: $(i+1)*n = (\log n + 1)*n$ -> $O(n \log n)$

THE VERY BAD CASE….

- If the array is inversely sorted.
- Let's see the problem:
  - **90 80 70 60 50 40 30 20 10 0**
- What happens after the partition? This:
  - **0 20 30 40 50 60 70 80 90 10**
- This is almost sorted, but the algorithm doesn't know it.
- It will then call itself on an array of zero size (the left subarray) and an array of n-1 size (the right subarray).
- Producing:
  - **0 10 30 40 50 60 70 80 90 20**

THE VERY BAD CASE…

- In the worst case, we partition every time into an array of 0 elements and an array of n-1 elements
- This yields $O(n^2)$ time:
  - First call: Partition n elements, n operations
  - Second calls: Partition 0 and n-1 elements, n-1 operations
  - Third calls: Partition 0 and n-2 elements, n-2 operations
  - Draw the tree

- Yielding:
  - Operations = $n + n-1 + n-2 + \ldots + 1 = n(n+1)/2 \rightarrow O(n^2)$

SUMMARY

- What caused the problem was “blindly” choosing the pivot from the right end.
- In the case of a reverse sorted array, this is not a good choice at all

- Can we improve our choice of the pivot?
  - Let’s choose the middle of three values

MEDIAN-OF-THREE PARTITIONING

- Everytime you partition, choose the median value of the left, center and right element as the pivot

- Example:
  - **44 11 55 33 77 22 00 99 101 66 88**

- Pivot: Take the median of the leftmost, middle and rightmost
  - **44 11 55 33 77 22 00 99 101 66 88** - Median: 44
- Then partition around this pivot:
  - **11 33 22 00 44 55 77 99 101 66 88**
- Increases the liklihood of an equal partition
  - Also, it cannot possibly be the worst case

HOW THIS FIXES THE WORST CASE?

- Here's our array:
  - 90 80 70 60 50 40 30 20 10 0

- Let's see on the board how this fixes things
- In fact in a perfectly reversed array, we choose the middle element as the pivot!
  - Which is optimal
  - We get $O(N \log N)$

- Vast majority of the time, if you use QuickSort with a Median-Of-Three partition, you get $O(N \log N)$ behavior

ONE FINAL OPTIMIZATION…

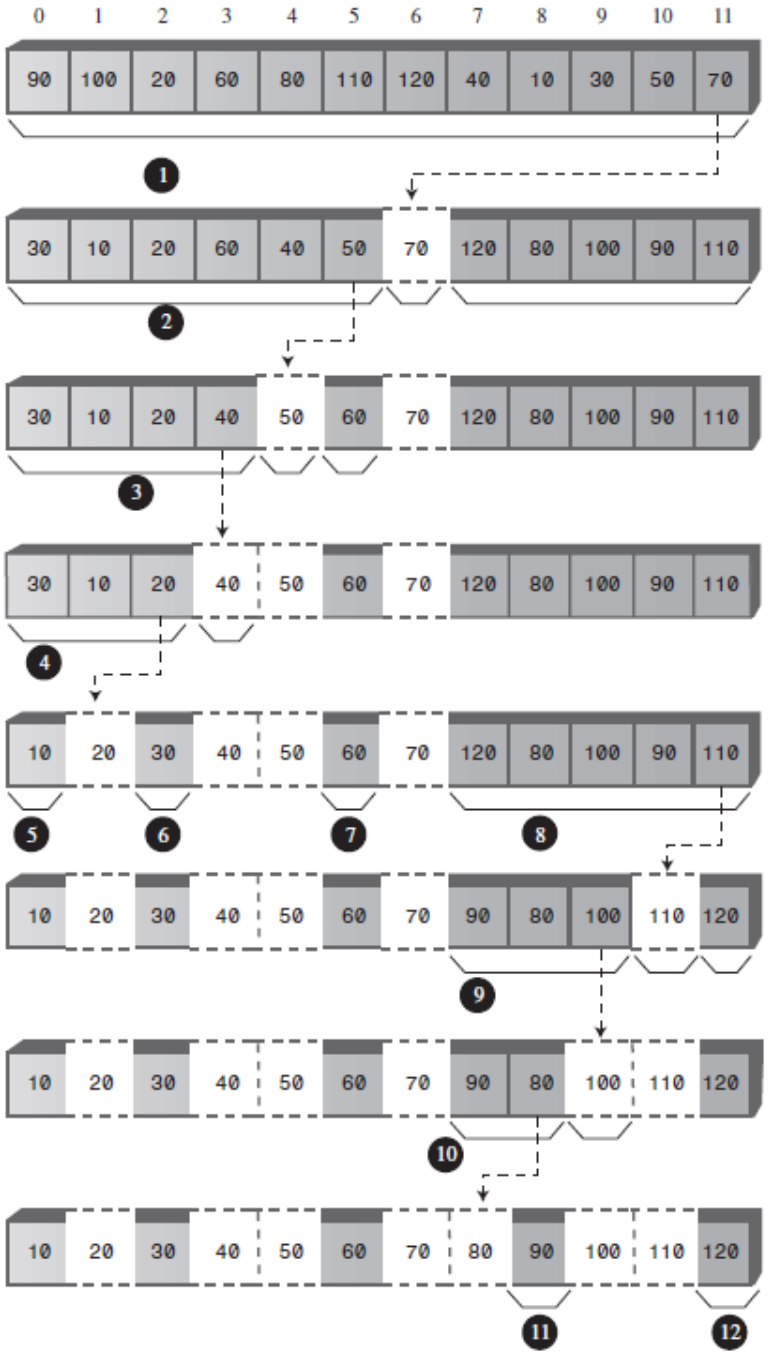
- After a certain point, just doing insertion sort is faster than partitioning small arrays and making recursive calls

- Once you get to a very small subarray, you can just sort with insertion sort
- You can experiment a bit with ‘cutoff’ values
  - Knuth: n=9

OPERATION COUNT ESTIMATES

- For QuickSort
- n=8: 30 comparisons, 12 swaps
- n=12: 50 comparisons, 21 swaps
- n=16: 72 comparisons, 32 swaps
- n=64: 396 comparisons, 192 swaps
- n=100: 678 comparisons, 332 swaps
- n=128: 910 comparisons, 448 swaps

- The only competitive algorithm is mergesort
  - But, takes much more memory like we said.



The quicksort process.

Summary of Quicksort

Quicksort operates in $O(N * \log N)$ time (except when the simpler version is applied to already-sorted data).

Subarrays smaller than a certain size (the cutoff) can be sorted by a method other than quicksort.

The insertion sort is commonly used to sort subarrays smaller than the cutoff.

The insertion sort can also be applied to the entire array, after it has been sorted down to a cutoff point by quicksort.

Swaps and Comparisons in Quicksort

N	8	12	16	64	100	128
$\log_2 N$	3	3.59	4	6	6.65	7
$N*\log_2 N$	24	43	64	384	665	896
Comparisons: $(N+2)*\log_2 N$	30	50	72	396	678	910
Swaps: fewer than $N/2*\log_2 N$	12	21	32	192	332	448

*The $\log_2 N$ quantity used in the table is true only in the best-case scenario, where each subarray is partitioned exactly in half. For random data, it is slightly greater.

OUTLINE

- Sorting
  - Simple Sorting
  - Advanced Sorting
- Hashing

HASH TABLES: OVERVIEW

- Provide very fast insertion and searching
  - Both are O(1)
  - Is this too good to be true?
- Disadvantages
  - Based on arrays, so the size must be known in advance
  - Performance degrades when the table becomes full
  - No convenient way to sort data
- Summary
  - Best structure if you have no need to visit items in order and you can predict the size of your database in advance.

MOTIVATION

- Let's suppose we want to insert a key into a data structure, where the key can fall into a range from *0* to *m*, where *m* is very large
- And we want to be able to find the key quickly

- Clearly, one option is to use an array of size *m+1*
  - Put each key into its corresponding slot (*0* to *m*)
  - Searching is then constant time

- But we may only have *n* keys, where *n* <<< *m*
  - So we waste a ton of space!

MOTIVATION

- Moreover, we may not be storing integers
- For example, if we store words of the dictionary
  - Our ‘range’ is ‘a’ to ‘zyzzyva’
  - And we’re talking hundreds of thousands of words in between

- So the ‘mapping’ is not necessarily clear
- For example, we won’t know immediately that the word ‘frog’ is at 85,467th word in the dictionary
  - BTW, I’m not claiming that is right. ☺

HASH TABLE

- Idea
  - Provide easy searching and insertion by mapping keys to positions in an array
  - This mapping is provided by a ***hash function***
    - Takes the key as input
    - Produces an index as output
- The array is called a ***hash table***.

HASH FUNCTION: EXAMPLE

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

- The easiest hash function is the following:
  - H(key) = key % tablesize
  - H(key) now contains a value between 0 and tablesize-1
- So if we inserted the following keys into a table of size 10: 13,11456, 2001, 157
  - You probably already see potential for collisions
  - Patience, we'll come to it!

WHAT HAVE WE ACCOMPLISHED?

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

- We have stored keys of an unpredictable large range into a smaller data structure
- And searching and inserting becomes easy!

- To find a key k, just retrieve table[$H(k)$]
- To insert a key k, just set table[$H(k)$] = k
- Both are O(1)!

WHAT'S OUR PRICE?

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

- Of course, you've probably already realized, multiple values in our range could map to the same hash table index
- For example, if we used:
  - H(k) = k % 10
- Then, tried to insert 207
  - H(207) = 7
- We have a ***collision*** at position 7

WHAT HAVE WE LEARNED?

- If we use hash tables, we need the following:
  - Some way of handling collisions. We'll study a couple ways:
    - Open addressing
      - Which has 3 kinds: linear probing, quadratic probing, and double hashing
    - Separate chaining

- Also, the choice of the hash function is delicate
  - We can produce hash functions which are more or less likely to have high collision frequencies
  - We'll look at potential options

LINEAR PROBING

- Presumably, you will have define your hash table size to be 'safe'
  - As in, larger than the maximum amount of items you expect to store
- As a result, there should be some available cells

- In ***linear probing***, if an insertion results in a collision, search sequentially until a vacant cell is found
  - Use wraparound if necessary

LINEAR PROBING: EXAMPLE

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

- Again, say we insert element 207
- H(207) = 207 % 10 = 7

- This results in a collision with element 157
- So we search linearly for the next available cell, which is at position 8
  - And put 207 there

LINEAR PROBING

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	426

- Note: This complicates insertion and searching a bit!
- For example, if we then inserted element 426, we would have to check three cells before finding a vacant one at position 9
- And searching, is not simply a matter of applying *H(k)*
  - You apply *H(k)*, and probe!

LINEAR PROBING: CLUSTERS

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	426

- As the table to the right illustrates, linear probing also tends to result in the formation of clusters.
  - Where large amounts of cells in a row are populated
  - And large amounts of cells are sparse
- This becomes worse as the table fills up
  - Degrades performance

LINEAR PROBING: CLUSTERS

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	426

- LaFore: A cluster is like a 'faint scene' at a mall
- Initially, the first arrivals come
  - Later arrivals come because they wonder why everyone was in one place
  - As the crowd gets bigger, more are attracted
- Same thing with clusters!
  - Items that hash to a value in the cluster will add to its size

LINEAR PROBING

- One option: If the table becomes full enough, double its size
- Note this is not quite as simple as it seems
  - Because for every value inside, you have to recompute its hash value
  - The hash function is necessarily different:
    - $H(k) = k \% 20$
  - But, less clustering

Index	Value
0	
1	2001
2	
3	
4	
5	
6	426
7	207
8	
9	
10	
11	
12	
13	13
14	
15	
16	11456
17	157
18	
19	
20	

QUADRATIC PROBING

- The main problem with linear probing was its potential for clustering
- Quadratic probing attempts to address this
  - Instead of linearly searching for these next available cell
    - i.e. for hash x, search cell x+1, x+2, x+3, x+4….
  - Search quadratically
    - i.e. for hash x, search cell x+1, x+4, x+9, x+16, x+25…

- Idea
  - On a collision, initially assume a small cluster and go to x+1
  - If that's occupied, assume a larger cluster and go to x+4
  - If that's occupied assume an even larger cluster, and go to x+9

QUADRATIC PROBING: EXAMPLE

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

- Returning to our old example with inserting 207
- $H(207) = 207 \% 10 = 7$
- This results in a collision with element 157
- In this case, slot 7 is occupied but slot 7+1=8 is open, so we put it there

QUADRATIC PROBING

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

- Now, if we insert 426
- H(426) = 426 % 10 = 6
  - Which is occupied
- Slot 6+1=7 is also occupied

- So we check slot:
  - 6+4=10
  - This passes the end, so we wraparound to slot 0 and insert there

QUADRATIC PROBING

Index	Value
0	426
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

- We have achieved a decrease in the cluster count
- Clusters will tend to be smaller and more sparse
  - Instead of having large clusters
  - And largely sparse areas
- Thus quadratic probing got rid of what we call ***primary clustering***.

QUADRATIC PROBING

Index	Value
0	426
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

- Quadratic probing does, however, suffer from ***secondary clustering***
- Where, if you have several keys hashing to the same value
  - The first collision requires one probe
  - The second requires four
  - The third requires nine
  - The fourth requires sixteen

QUADRATIC PROBING

Index	Value
0	426
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	207
9	

- Secondary clustering would happen if we inserted for example:
  - 827, 10857, 707 1117
  - Because they all hash to 7
- Not as serious a problem as primary clustering
- But there is a better solution that avoids both.

DOUBLE HASHING

- The problem thus far is that the probe sequences are always the same
  - For example: linear probing always generates x+1, x+2, x+3...
  - Quadratic probing always generates x+1, x+4, x+9…
- Solution: Make both the hash location and the probe dependent upon the key
  - Hash the key once to get the location
  - Hash the key a second time to get the probe
- This is called ***double hashing***.

SECOND HASH FUNCTION

- Characteristics of the hash function for the probe
  - It cannot be the same as the first hash function
  - It can NEVER hash to zero
    - Why not?

- Experts have discovered, this type of hash function works good for the probe:
  - $probe = c - (key \% c)$

  - Where c is a prime number that is smaller than the array size

DOUBLE HASHING: EXAMPLE

Index	Value
0	
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

- Returning to our old example with inserting 207
- *H(207) = 207 % 10 = 7*

- This results in a collision with element 157
- So we hash again, to get the probe
  - Suppose we choose *c=5*
  - Then:
    - *P(207) = 5 – (207 % 5)*
    - *P(207) = 5 – 2 = 3*

DOUBLE HASHING: EXAMPLE

Index	Value
0	207
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

- So we insert 207 at position:
  - *H(207) + P(207) =*
  - *7+3 =*
  - *10*
- Wrapping around, this will put 207 at position 0

DOUBLE HASHING: EXAMPLE

Index	Value
0	207
1	2001
2	
3	13
4	
5	
6	11456
7	157
8	
9	

- Now, let's again insert value 426
- We run the initial hash:
  - H(426) = 426 % 10 = 6
- We get a collision, so we probe:
  - P(426) = 5 – (426 % 5)
  - = 5 – 1 = 4
- And insert at location:
  - H(426) + P(426) = 10
  - Wrapping around, we get 0. Another collision!

DOUBLE HASHING: EXAMPLE

Index	Value
0	207
1	2001
2	
3	13
4	426
5	
6	11456
7	157
8	
9	

- So, we probe again
  - $P(426) = 4$
- So we insert at location $0+4 = 4$, and this time there is no collision

- Double hashing will in general produce the fewest clusters
  - Because both the hash and probe are key-dependent

NOTE…

- What is a potential problem with choosing a hash table of size 10 and a c of 5 for the probe, as we just did?
- Suppose we had a value k where H(k) = 0 and P(k) = 5
  - i.e., k = 0
- What would the probe sequence be?
- What's the problem?

PROBE SEQUENCE

- The probe sequence may never find an open cell!
- Because H(0) = 0, we'll start at hash location 0
  - If we have a collision, P(0) = 5 so we'll next check 0+5=5
  - If we have a collision there, we'll next check 5+5=10, with wraparound we get 0
  - We'll infinitely check 0 and 5, and never find an open cell!

DOUBLE HASHING REQUIREMENT

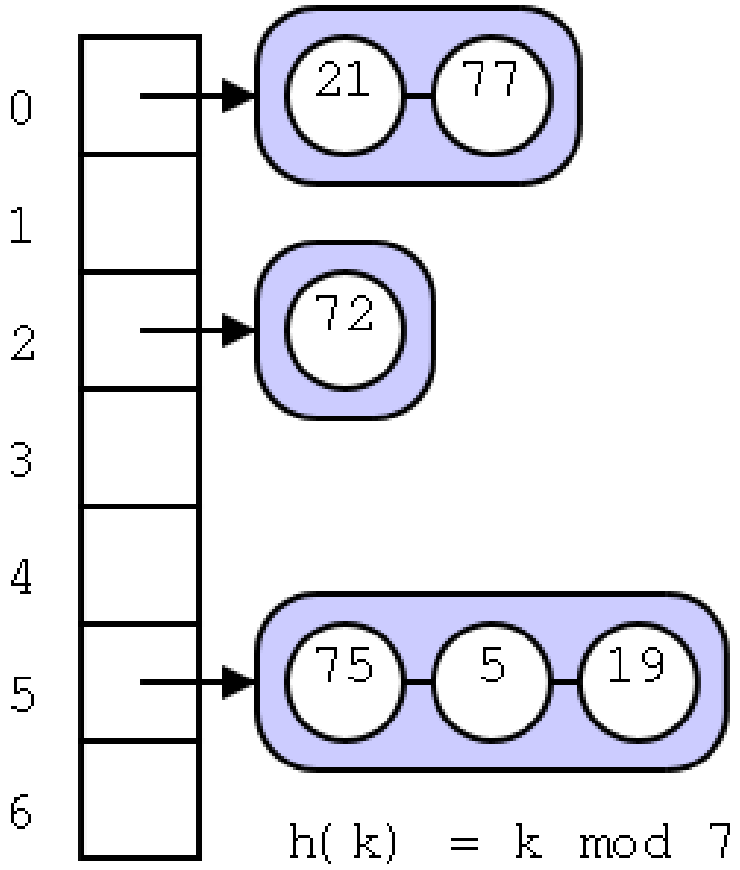
- The root of the problem is that the table size is not prime!
  - For example if the size were 11:
  - 0, 5, 10, 4, 9, 3, 8, 2, 7, 1, 6
  - If there is even one open cell, the probing is guaranteed to find it

- Thus, very important – ***a requirement of double hashing is that the table size is prime***.
  - So our previous table size of 10 is not a good idea
  - We would want 11, or 13, etc.

- Generally, for open addressing, double hashing is best

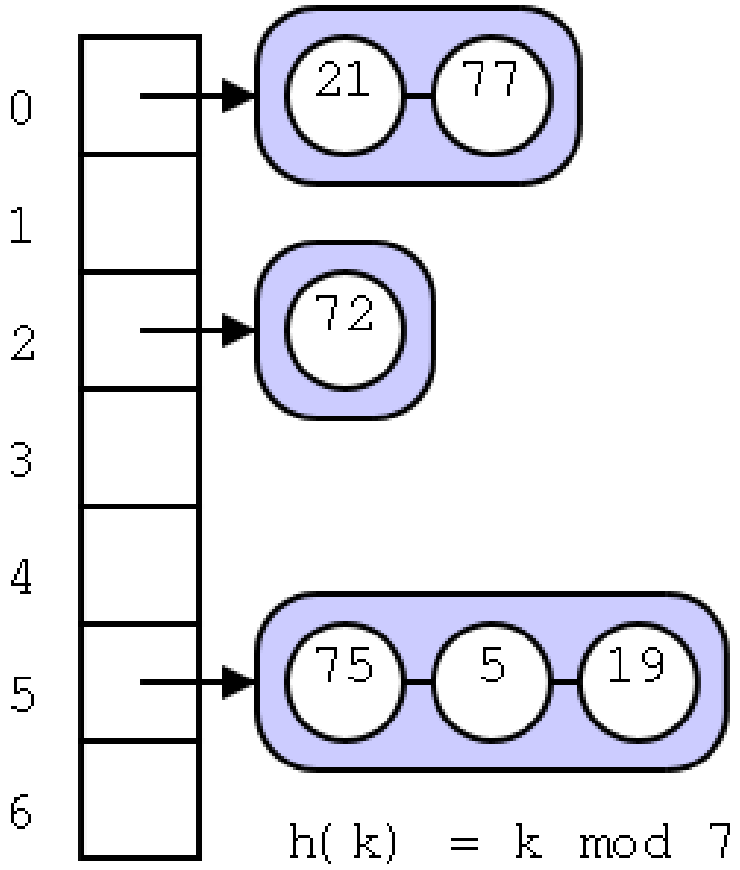
SEPARATE CHAINING

- The alternative to open addressing
- Does not involve probing to different locations in the hash table
- Rather, every location in the hash table contains a linked list of keys



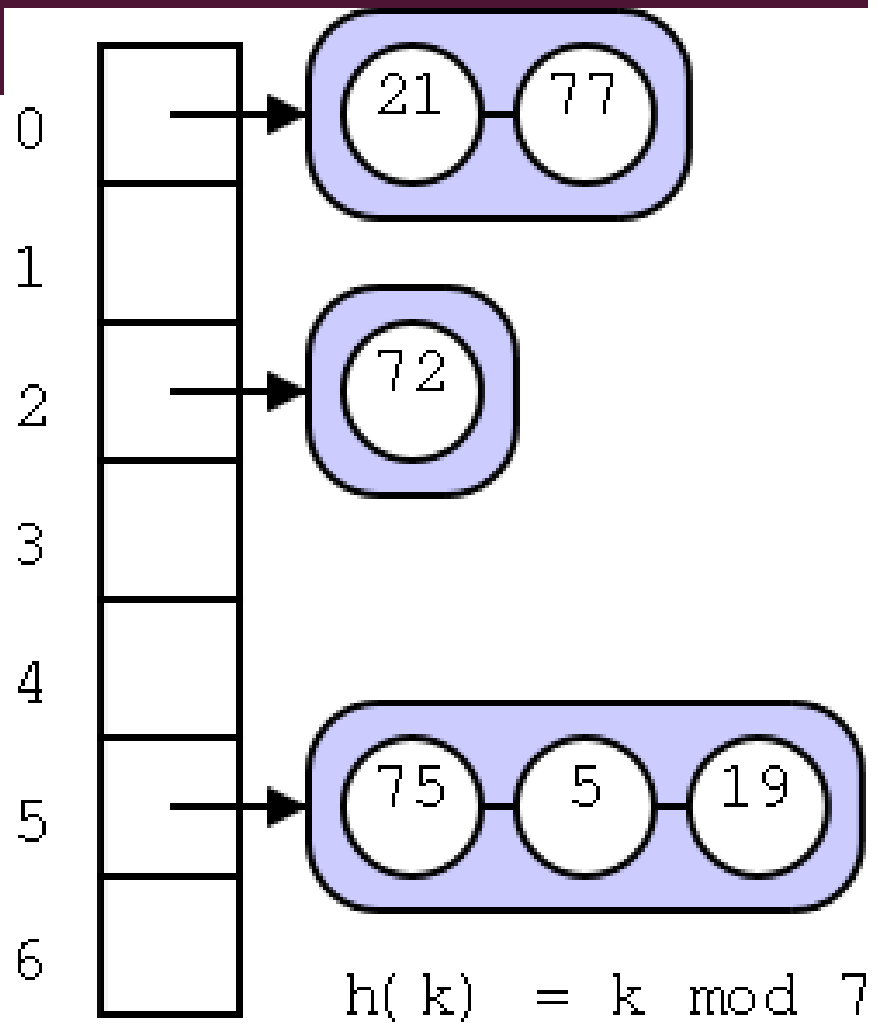
SEPARATE CHAINING

- Simple case, 7 element hash table
- H(k) = k % 7
- So:
  - 21, 77 each hash to location 0
  - 72 hashes to location 2
  - 75, 5, 19 hash to location 5
- Each is simply appended to the correct linked list



SEPARATE CHAINING

- In separate chaining, trouble happens when a list gets too full
- Generally, we want to keep the size of the biggest list, call it M, much smaller than N
  - Searching and insertion will then take O(M) time in the worst case



A GOOD HASH FUNCTION

- Has two properties:
  - Is computable quickly; so as not to degrade performance of insertion and searching
  - Can take a range of key values and transform them into indices such that the key values are distributed randomly across the hash table

- For random keys, the modulo (%) operator is good
- It is not always an easy task!

FOR EXAMPLE…

- Data can be highly non-random
- For example, a car-part ID:
  - 033-400-03-94-05-0-535

- For each set of digits, there can be a unique range or set of values!
  - i.e. Digits 3-5 could be a category code, where the only acceptable values are 100, 150, 200, 250, up to 850.
  - Digits 6-7 could be a month of introduction (0-12)
  - Digit 12 could be "yes" or "no" (0 or 1)
  - Digits 13-15 could be a checksum, a function of all the other digits in the code

RULE #1:
DON'T USE NON-DATA

- Compress the key fields down enough until every bit counts
- For example:
  - The category (bits 3-5, with restricted values 100, 150, 200, … , 850) counting by 50s needs to be compressed down to run from 0 to 15
  - The checksum is not necessary, and should be removed. It is a function of the rest of the code and thus redundant with respect to the hash table

RULE #2: USE ALL OF THE DATA

- Every part of the key should contribute to the hash function
- More data portions that contribute to the key, more likely it will be that the keys hash evenly
  - Saving collisions, which cause trouble no matter what the algorithm you use

RULE #3: USE A PRIME NUMBER FOR MODULO BASE

- This is a requirement for double hashing
- Important for quadratic probing
- Especially important if the keys may not be randomly distributed
  - The more keys that share a divisor with the array size, the more collisions
  - Example, non-random data which are multiples of 50
    - If the table size is 50, they all hash to the same spot
    - If the table size is 10, they all hash to the same pot
    - If the table size is 53, no keys divide evenly into the table size. Better!

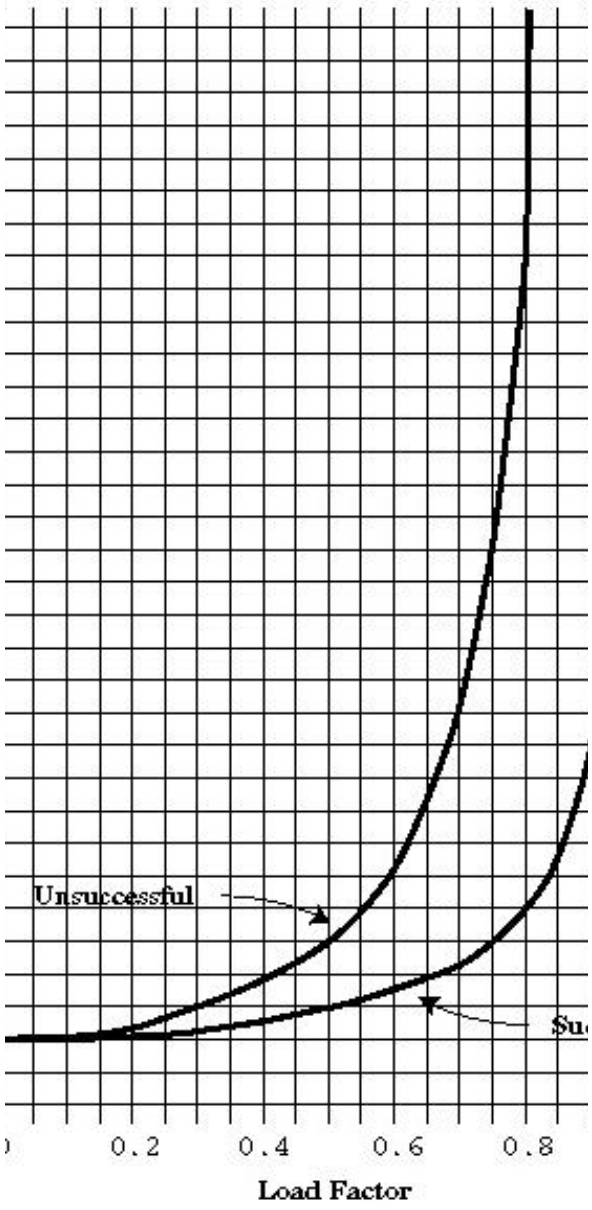
HASHING EFFICIENCY

- Insertion and Searching are O(1) in the best case
  - This implies no collisions
  - If you minimize collisions, you can approach this runtime

- If collisions occur:
  - Access times depend on resulting probe lengths
  - Every probe equals one more access
  - So every worst case insertion or search time is proportional to:
    - The number of required probes if you use open addressing
    - The number of links in the longest list it you use separate chaining

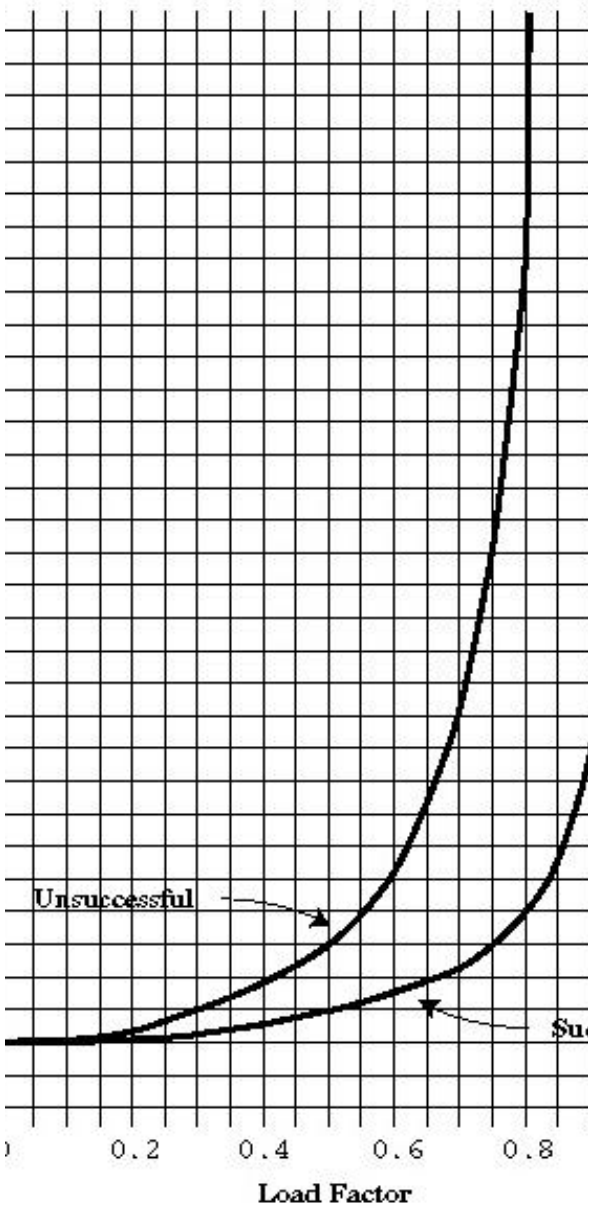
EFFICIENCY: LINEAR PROBING

- Let's assume a load factor *L*, where *L* is the percentage of hash table slots which are occupied.
- Knuth showed that, for a successful search:
  - $P = (1 + 1 / (1 - L)^2) / 2$
- For an unsuccessful search:
  - $P = (1 + 1 / (1 - L)) / 2$



EFFICIENCY: LINEAR PROBING

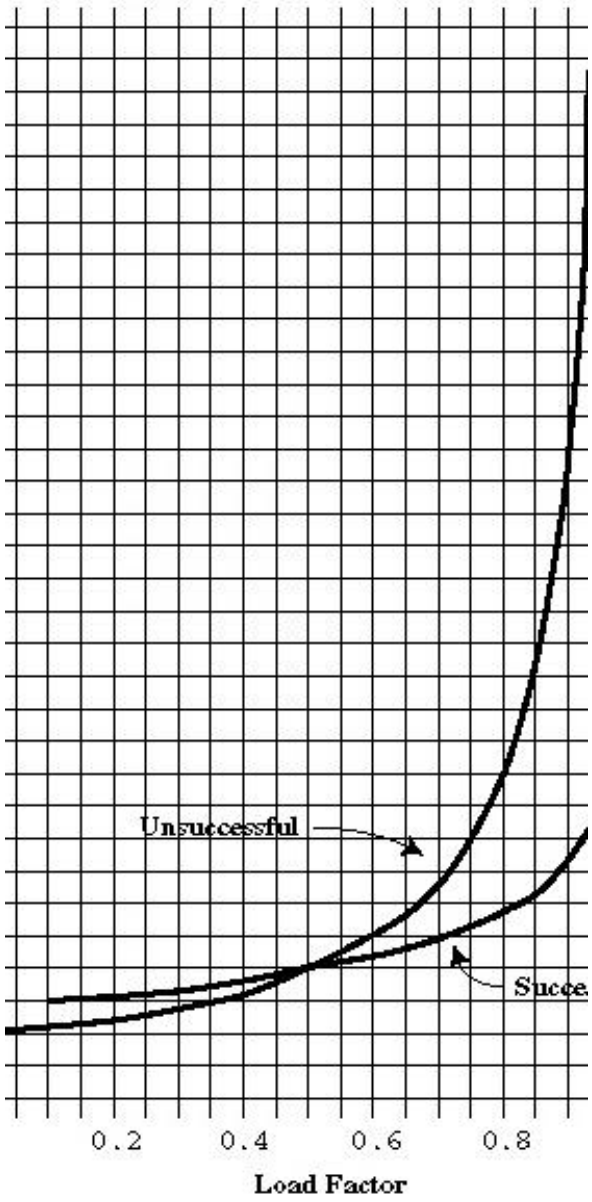
- What's the ideal load factor?
- At L=0.5:
  - Successful search takes 1.5 probes
  - Unsuccessful takes 2.5
- At L = 2/3:
  - Successful: 2.0
  - Unsuccessful: 5.0
- Good to keep load factor under 0.5!



EFFICIENCY: QUADRATIC PROBING AND DOUBLE HASHING

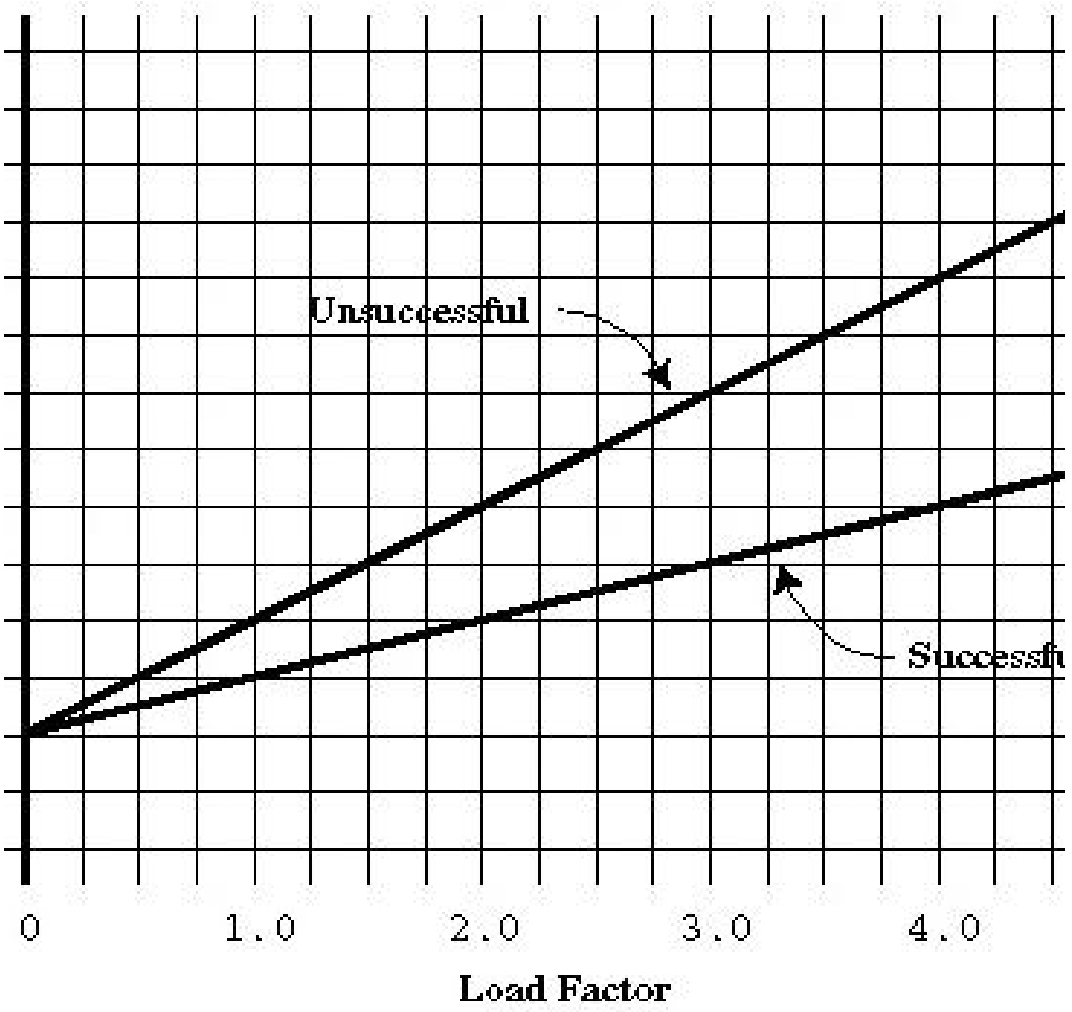
- Again, assume a load factor L, where L is the percentage of hash table slots which are occupied.

- Knuth showed that, for a successful search:
  - P = 1 / (1 – L)
- For an unsuccessful search:
  - P = -log(1-L) / L
- Can tolerate somewhat higher L



EFFICIENCY: SEPARATE CHAINING

- Here L is a bit more complicated:
  - For N elements
  - And an array of size S
  - L = N / S
- Successful (average):
  - 1 + (L/2)
- Unsuccessful:
  - 1 + L



SUMMARY: WHEN TO USE WHAT

- If the number of items that will be inserted is uncertain, use separate chaining
  - Must create a LinkedList class
  - But performance degrades only linearly
  - With open addressing, major penalties
- Otherwise, use double hashing, unless…
  - Plenty of memory available
  - Low load factors
  - Then linear or quadratic probing should be done for ease of implementation

THANKS