Sorting and Searching

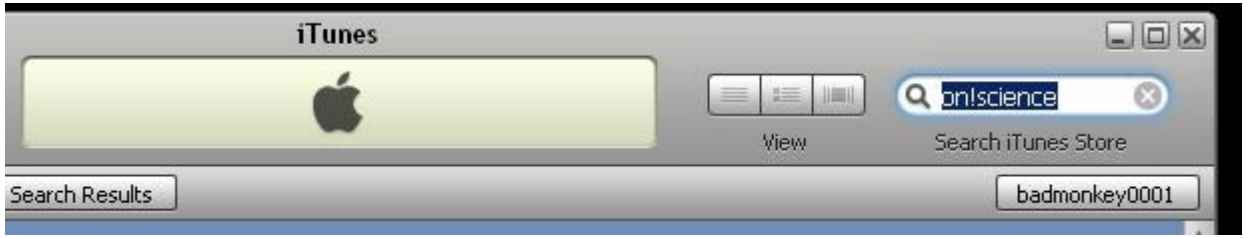
"There's nothing in your head the sorting hat can't see. So try me on and I will tell you where you ought to be."

-The Sorting Hat, *Harry Potter and the Sorcerer's Stone*

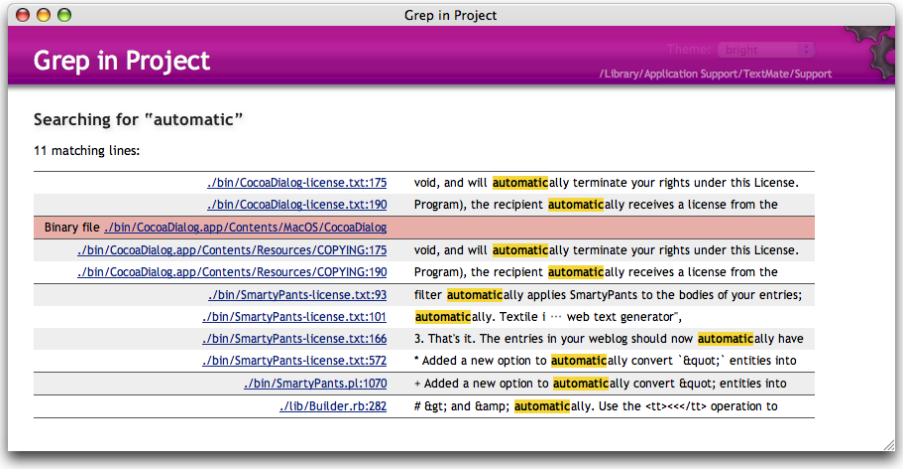
Sorting and Searching

- Fundamental problems in computer science and programming
- Sorting done to make searching easier
- Multiple algorithms to solve the same problem
  - How do we know which algorithm is "better"?
- Look at searching first
- Examples will use arrays of ***ints*** to illustrate algorithms

Searching


iTunes
bn!science
View
Search iTunes Store
Search Results
badmonkey0001


Google™
recursive backtracking
Google Search
I'm Feeling Lucky
Advanced Search
Preferences
Language Tools


Grep in Project
Grep in Project
/Library/Application Support/TextMate/Support
Searching for "automatic"
11 matching lines:
./bin/CocoaDialog-license.txt:175 void, and will automatically terminate your rights under this License.
./bin/CocoaDialog-license.txt:190 Program), the recipient automatically receives a license from the
Binary file ./bin/CocoaDialog.app/Contents/MacOS/CocoaDialog
./bin/CocoaDialog.app/Contents/Resources/COPYING:175 void, and will automatically terminate your rights under this License.
./bin/CocoaDialog.app/Contents/Resources/COPYING:190 Program), the recipient automatically receives a license from the
./bin/SmartyPants-license.txt:93 filter automatically applies SmartyPants to the bodies of your entries;
./bin/SmartyPants-license.txt:101 automatically. Textile i ··· web text generator",
./bin/SmartyPants-license.txt:166 3. That's it. The entries in your weblog should now automatically have
./bin/SmartyPants-license.txt:572 * Added a new option to automatically convert `&quot;` entities into
./bin/SmartyPants.pl:1070 + Added a new option to automatically convert " entities into
./lib/Builder.rb:282 # > and & automatically. Use the <tt><<</tt> operation to

Searching

- Given a list of data, find the location of a particular value or report that value is not present
- Linear search
  - intuitive approach:
    - start at first item
    - is it the one I am looking for?
    - If not, go to next item
    - repeat until found or all items checked
- If items not sorted or unsortable this approach is necessary

Linear Search

```
/*      pre: list != null
        post: return the index of the first occurrence
        of target in list or -1 if target not present in
        list
*/
public int linearSearch(int list[], int target)
{
    int i = ;
    while(i < list.length && list[i] != target)
       i++;

    if(i > list.length)
       return -1;
    else
      return i;
}
```

Question 1

- What is the average case Big-O of **linear search** in an array with *n* items, if an item is present?

A. O(n)

B. $O(n^2)$

C. O(1)

D. O(log(n))

E. O(n log(n))

Question 1

- What is the average case Big-O of **linear search** in an array with *n* items, if an item is present?

A. O(n)

B. $O(n^2)$

C. O(1)

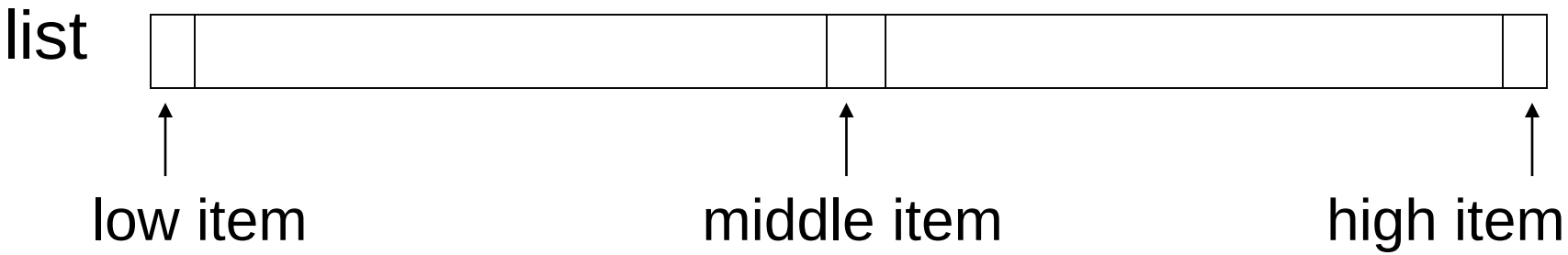
D. O(log(n))

E. O(n log(n))

Searching in a Sorted List

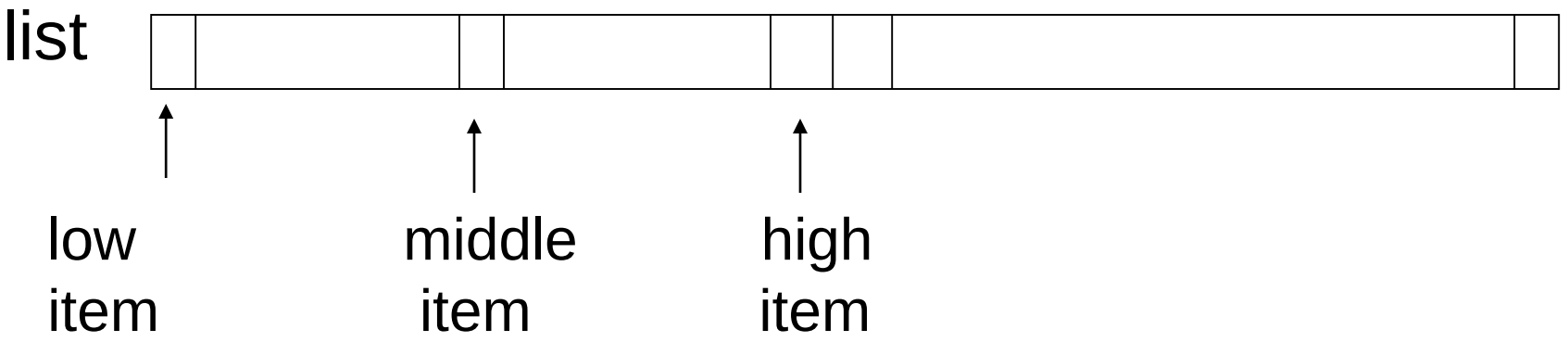
- If items are sorted, we can *divide and conquer*
- Dividing your work in half with each step
  - Generally a good thing
- The Binary Search on List in Ascending order
  - start at middle of list
    - is that the item?
    - if not, is it less than or greater than the item?
      - less than, move to second half of list
      - greater than, move to first half of list
  - repeat until found or sub list size = 0

Binary Search



Is middle item what we are looking for?

If not, is it more or less than the target? (Assume lower)



and so forth…

Recursive Binary Search

```
public static int search(int list[], int target)
{
    return b-search(list, target, 0, list.length – 1);
}

public static int b-search(int list[], int target,
                                    int low, int high)
{
    if( low <= high ){
        int mid = low + ((high - low) / 2);
         if( list[mid] == target )
            return mid;
        else if( list[mid] > target )
            return b-search(list, target, low, mid – 1);
        else
            return b-search(list, target, mid + 1, high);
    }
    return -1;
}
```

Question 2

What is the worst case Big O of **binary search** in an array with *n* items, if an item is present?

A. O(n)

B. $O(n^2)$

C. O(1)

D. O(log(n))

E. O(n log(n))

Question 2

What is the worst case Big O of **binary search** in an array with *n* items, if an item is present?

A. O(n)

B. $O(n^2)$

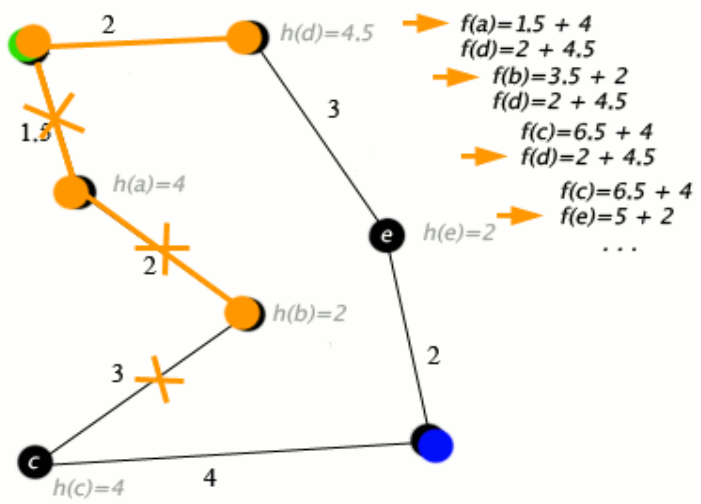
C. O(1)

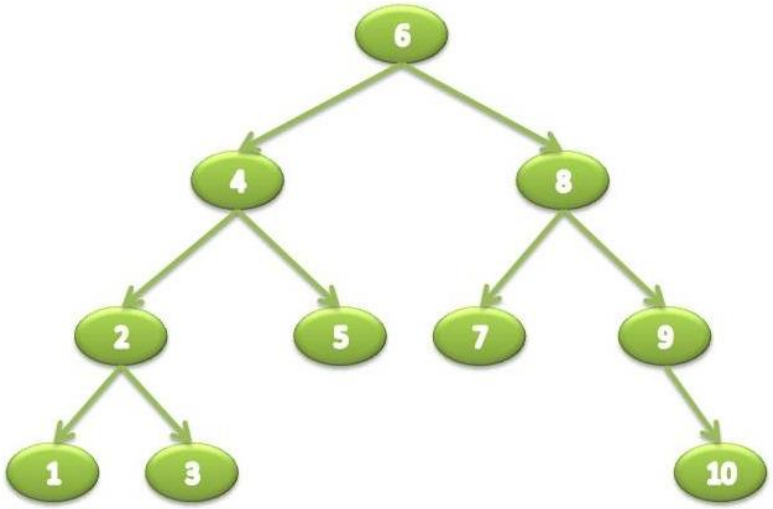
D. O(log(n))

E. O(n log(n))

Other Searching Algorithms

- Interpolation Search
  - more like what people really do
- Binary Search Trees
- Hash Table Searching
- Best-First
- A*





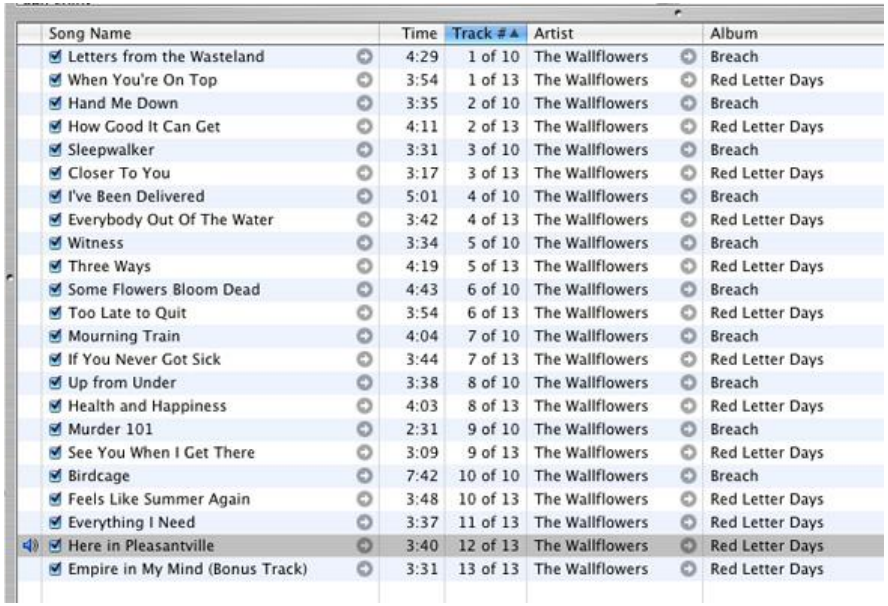
Sorting

U.S. All-time List - Marathon

As of 4/24/08

Women

1	1	2:19:36	Deena Kastor nee Drossin
2		2:21:16	Drossin (2)
3	2	2:21:21	Joan Benoit Samuelson
4		2:21:25	Kastor (3)
5		2:22:43a	Benoit (2)
6		2:24:52a	Benoit (3)
7		2:26:11	Benoit (4)
8	3	2:26:26a	Julie Brown
9	4	2:26:40a	Kim Jones



Song Name	Time	Track #	Artist	Album
Letters from the Wasteland	4:29	1 of 10	The Wallflowers	Breach
When You're On Top	3:54	1 of 13	The Wallflowers	Red Letter Days
Hand Me Down	3:35	2 of 10	The Wallflowers	Breach
How Good It Can Get	4:11	2 of 13	The Wallflowers	Red Letter Days
Sleepwalker	3:31	3 of 10	The Wallflowers	Breach
Closer To You	3:17	3 of 13	The Wallflowers	Red Letter Days
I've Been Delivered	5:01	4 of 10	The Wallflowers	Breach
Everybody Out Of The Water	3:42	4 of 13	The Wallflowers	Red Letter Days
Witness	3:34	5 of 10	The Wallflowers	Breach
Three Ways	4:19	5 of 13	The Wallflowers	Red Letter Days
Some Flowers Bloom Dead	4:43	6 of 10	The Wallflowers	Breach
Too Late to Quit	3:54	6 of 13	The Wallflowers	Red Letter Days
Mourning Train	4:04	7 of 10	The Wallflowers	Breach
If You Never Got Sick	3:44	7 of 13	The Wallflowers	Red Letter Days
Up from Under	3:38	8 of 10	The Wallflowers	Breach
Health and Happiness	4:03	8 of 13	The Wallflowers	Red Letter Days
Murder 101	2:31	9 of 10	The Wallflowers	Breach
See You When I Get There	3:09	9 of 13	The Wallflowers	Red Letter Days
Birdcage	7:42	10 of 10	The Wallflowers	Breach
Feels Like Summer Again	3:48	10 of 13	The Wallflowers	Red Letter Days
Everything I Need	3:37	11 of 13	The Wallflowers	Red Letter Days
Here in Pleasantville	3:40	12 of 13	The Wallflowers	Red Letter Days
Empire in My Mind (Bonus Track)	3:31	13 of 13	The Wallflowers	Red Letter Days

Sorting Fun:
Why Not Bubble Sort?

Sorting

- A fundamental application for computers
- Done to make finding data (searching) faster
- Many different algorithms for sorting
- One of the difficulties with sorting is working with a fixed size storage container (array)
  - if resize, that is expensive (slow)
- The "simple" sorts run in quadratic time $O(n^2)$
  - selection sort
  - insertion sort
  - bubble sort

Selection Sort

- Algorithm
  - Search through the list and find the smallest element
  - Swap the smallest element with the first element
  - Repeat: find the second smallest element starting at second element

```
public static void selectionSort(int list[])
{       int min;
        int temp;
        for(int i = 0; i < list.length - 1; i++) {
           min = i;
           for(int j = i + 1; j < list.length; j++) {
               if( list[j] < list[min] )
                  min = j;
               temp = list[i];
               list[i] = list[min];
               list[min] = temp;
            }
      }
}
```

Selection Sort in Practice

44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

What is the *actual* number of statements executed of the selection sort code, given a list of *n* elements? What is the Big O?

Selection Sort Code

```
public static void selectionSort(int list[]){
      int min;
      int temp;
      for(int i = 0; i < list.length - 1; i++){
         min = i;
         for(int j = i + 1; j < list.length; j++){
                  if( list[j] < list[min] )
                     min = j;
                  temp = list[i];
                  list[i] = list[min];
                  list[min] = temp;
            }
      }
}
```

Insertion Sort

- Another of the $O(n^2)$ sorts:
  - Start with first item, assume it's sorted
  - Compare the second item to the first
    - if it's smaller, swap
  - Compare the third item to the second
    - If smaller, swap
    - Compare again with first, if smaller swap again
- And so forth…

Insertion Sort in Practice

44 68 191 119 119 37 83 82 191 45 158 130 76 153 39 25

What is the *actual* number of statements executed of the selection sort code, given a list of *n* elements? What is the Big O?

Insertion Sort Code

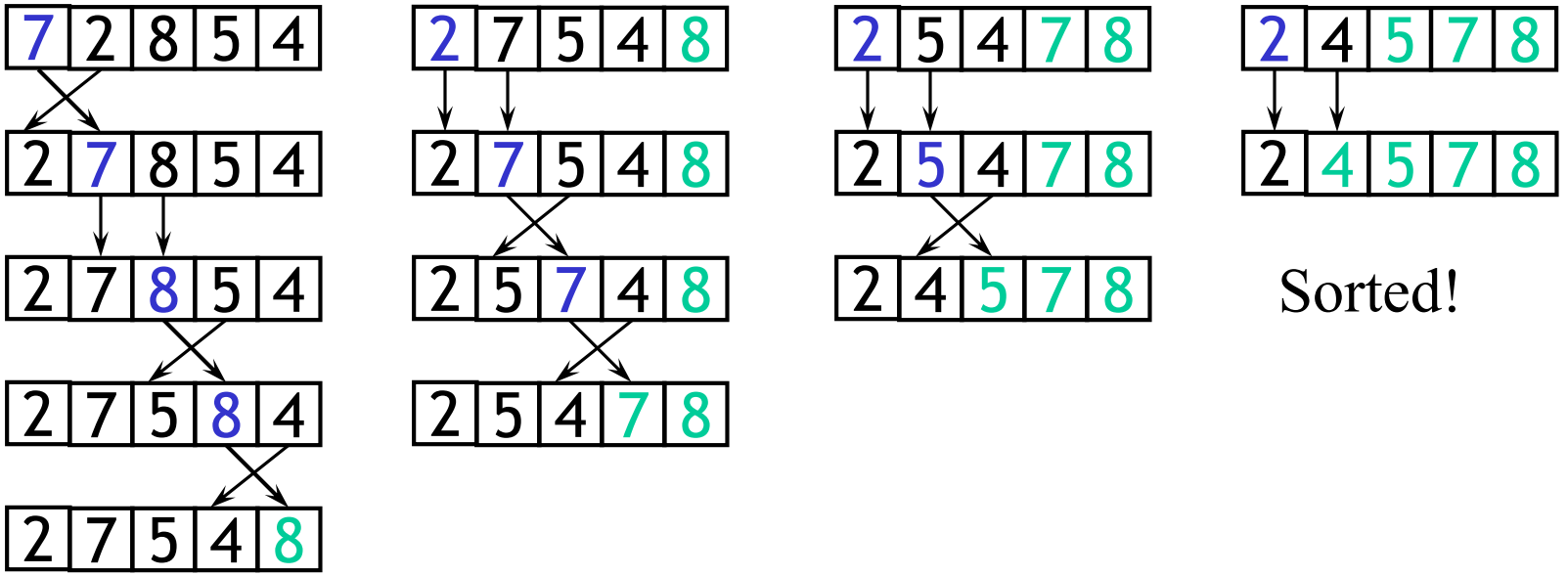
```
public void insertionSort(int list[])
{     int temp, j;
      for(int i = 1; i < list.length; i++)
      {     temp = list[i];
            j = i;
            while( j > 0 && temp < list[j - 1])
            {     // swap elements
                  list[j] = list[j - 1];
                  list[j - 1] = temp;
                  j--;
            }
      }
}
```

Big O?

Bubble Sort

- Start at the beginning of the data set:
  - Compare the first two elements
    - if the first is greater than the second, swap them.
  - Compare second to the third
    - if the second is greater than the third, swap them.
  - Continue doing this for each pair of adjacent elements to the end of the data set.
- Start again with the first two elements, repeating until no swaps have occurred on the last pass.

Example of Bubble Sort



Bubble Sort Code

```
public static void BubbleSort( int[] num )
{
     int j;
     boolean flag = true; // set flag to true for first pass
     int temp;            //holding variable
     while ( flag )
     {
            flag= false;    //set flag to false for possible swap
            for( j=0;  j < num.length -1;  j++ )
            {
                   if ( num[ j ] > num[j+1] )
                   {
                                //swap elements
                          temp = num[ j ];
                          num[ j ] = num[ j+1 ];
                          num[ j+1 ] = temp;
                         flag = true;   //shows a swap occurred
                   }
            }
      }
}
```

Big O?

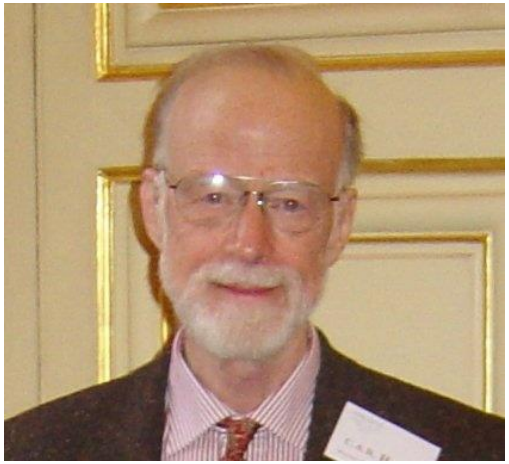
Comparing Algorithms

- Which algorithm do you think will be faster given random data:
  - selection sort?
  - bubble sort?
  - insertion sort?
- Why?

Sub-Quadratic Sorting Algorithms

Sub-Quadratic = Big O better than $O(n^2)$

Quicksort



- Invented by C.A.R. (Tony) Hoare
- A Divide-and-Conquer approach that uses recursion:
  - If the list has 0 or 1 elements, it's sorted
  - Otherwise, pick any element ***p*** in the list. This is called the ***pivot value***
  - Partition the list, minus the pivot, into two sub lists:
    - values less than or greater than the pivot.
    - equal values go to either
  - Return the Quicksort of the first list followed by the Quicksort of the second list.

Quicksort in Action

39 23 17 90 33 72 46 79 11 52 64 5 71

Pick middle element as pivot: 46

Partition list:

23 17 5 33 39 11 46 79 72 52 64 90 71

- quicksort the less than list

Pick middle element as pivot: 33

23 17 5 11 33 39

-quicksort the less than list, pivot now 5

{} 5 23 17 11

- quicksort the less than list, base case

- quicksort the greater than list

Pick middle element as pivot: 17

and so on….

Quicksort on Another Data Set

44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25

```
public void quicksort(Comparable list[], int lo, int hi)
{
      int p = partition(list, lo, hi);
      quicksort(list, lo, p - 1);
      quicksort(list, p + 1, hi);
}

public static void swap(Object a[], int index1, int index2)
{
      Object tmp = a[index1];
      a[index1] = a[index2];
      a[index2] = tmp;
}
```

```
public int partition(Comparable list[], int lo, int hi)
{
        // pivot at start position
        int pivot = list[lo];
        // keep track of last value <= pivot
        int i = lo;
        // move smaller values down in list
        for(int j = lo + 1; j <= hi)
        {
                if(list[j] <= pivot)
                {
                        i++;
                        swap(list[i], list[j]);
                }
        }
        // put pivot in correct position in partitioned list
        swap(list[i], list[lo]);
        // index of pivot value
        return i;
}
```

Question 3

- What is the Big-O of Quicksort?

A. $O(2^n)$
B. $O(n^2)$
C. O(n log(n))
D. O(n)
E. O(1)

Question 3

- What is the Big-O of Quicksort?

A. $O(2^n)$

B. $O(n^2)$

C. O(n log(n))

D. O(n)

E. O(1)

Quicksort Caveats

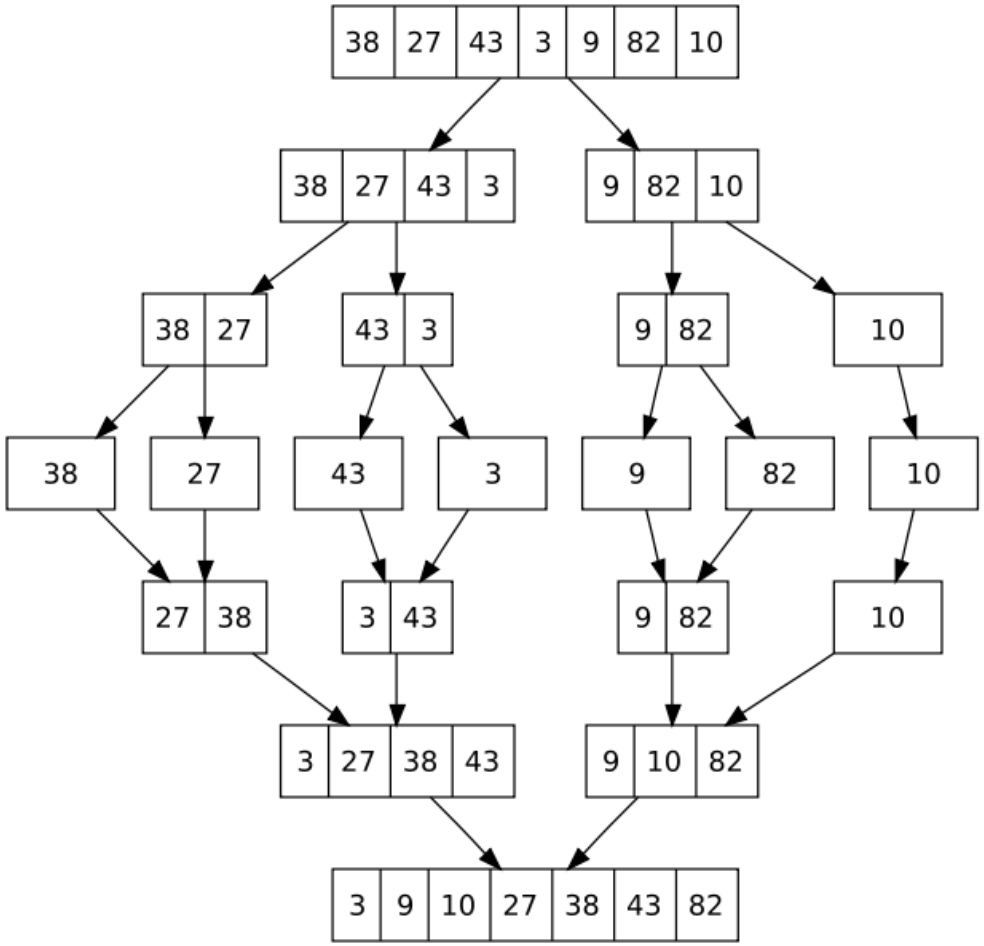
- Coding the partition step is usually the hardest part
  - Good partitions and bad partitions
    - Depends on underlying data

Mergesort Algorithm

Don Knuth cites John von Neumann as the creator of this algorithm:

- If a list has 1 element or 0 elements, it's sorted
- If a list has more than 1, split into two separate lists
- Perform this algorithm on each of those smaller lists
- Take the 2 sorted lists and merge them together

Mergesort



When implementing, one temporary array is used instead of multiple temporary arrays.

Why?

Mergesort Code

```
/**
 * perform a merge sort on the data in c
 * @param c c != null, all elements of c
 * are the same data type
 */
public static void sort(Comparable c[])
{       Comparable temp[] = new Comparable[ c.length ];
        mergeSort(c, temp, 0, c.length - 1);
}

private static void mergeSort(Comparable list[],
                Comparable temp[], int low, int high)
{       if( low < high){
                int center = (low + high) / 2;
                mergeSort(list, temp, low, center);
                mergeSort(list, temp, center + 1, high);
                merge(list, temp, low, center + 1, high);
        }
}
```

Merge Code

```
private static void merge( Comparable list[], Comparable temp[], int leftPos, int rightPos, int rightEnd)
{
            int leftEnd = rightPos - 1;
            int tempPos = leftPos;
            int numElements = rightEnd - leftPos + 1;
            //main loop
            while( leftPos <= leftEnd && rightPos <= rightEnd){
                        if( list[ leftPos ].compareTo(list[rightPos]) <= 0)
                        {
                                    temp[ tempPos ] = list[ leftPos ];
                                    leftPos++;
                        }
                        else
                        {
                                    temp[ tempPos ] = list[ rightPos ];
                                    rightPos++;
                        }
                        tempPos++;
            }
            //copy rest of left half
            while( leftPos <= leftEnd)
            {
                        temp[ tempPos ] = list[ leftPos ];
                        tempPos++;
                        leftPos++;
            }
            //copy rest of right half
            while( rightPos <= rightEnd)
            {
                        temp[ tempPos ] = list[ rightPos ];
                        tempPos++;
                        rightPos++;
            }
            //Copy temp back into list
            for(int i = 0; i < numElements; i++, rightEnd--)
                        list[ rightEnd ] = temp[ rightEnd ];
}
```

Can We Do Better?

Yes, sort of.

Time complexity of Sorting

- Of the several sorting algorithms discussed so far, the best ones:
  - Mergesort and Quicksort (best one in practice): O(n log(n)) on average, $O(n^2)$ worst case
- Can we do better than O(n log(n))?
  - No.
  - Proven:
    - any comparison-based sorting algorithm will need at least O(n log(n)) operations

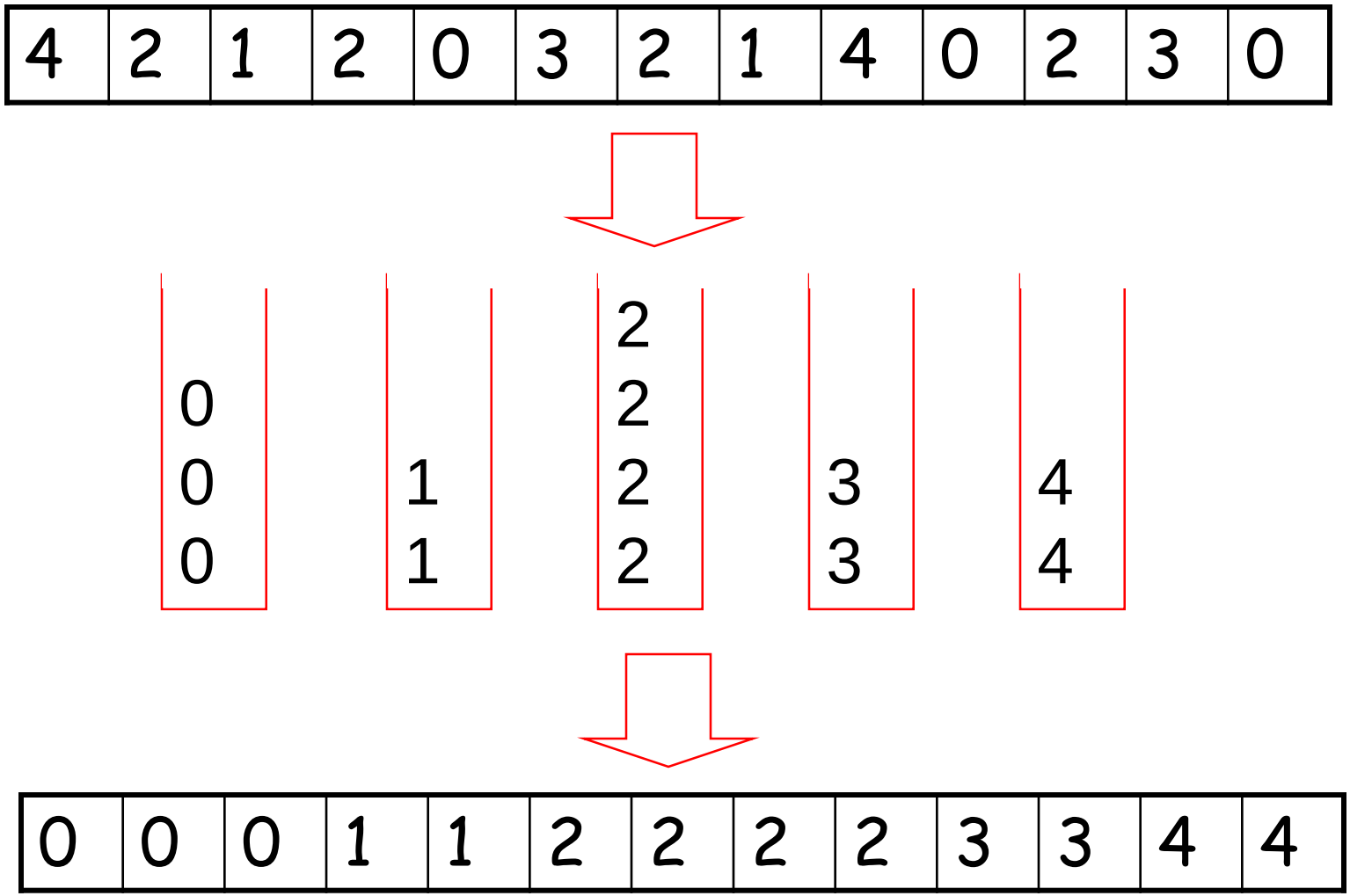
Restrictions on the problem

- Suppose:
  - the values repeat
  - but the values have a limit (digits from 0 to 9)
- Sorting should be easier
- Is it possible to come up with an algorithm better than O(n log(n))?
  - Yes
  - Strategy will not involve comparisons

Bucket Sort

- Suppose the values are in the range 0..m-1
  - start with m empty *buckets* numbered 0 to m-1
  - scan the list
  - place element ***i*** in bucket ***i***
  - output the buckets in order
- Need:
  - an array of buckets
  - values to be sorted will be the indexes to the buckets
- No comparisons will be necessary

Example



Values versus Entries

- If sorting values, each bucket is just a counter that we increment whenever a value matching the bucket's number is encountered
- If we were sorting entries according to keys, then each bucket is a queue
  - Entries are enqueued into a matching bucket
  - Entries will be dequeued back into the array after the scan

Time Complexity

- Bucket initialization: O(m)
- From array to buckets: O(n)
- From buckets to array: O(n)
  - this stage is a nested loop but only dequeue from each bucket until they are all empty
    - ***n*** dequeue operations in all

- Since ***m*** likely small compared to ***n***, Bucket sort is O(n)
  - Strictly speaking, time complexity is O (n + m)

Sorting Integers

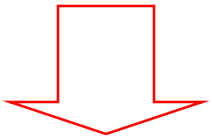
- Can we perform Bucket sort on any array of positive integers?
  - Yes, but the number of buckets depends on the maximum integer value
  - If you are sorting 1000 integers and the maximum value is 999999, you will need 1 million buckets!
  - Therefore, time complexity is not really O(n) because
    - ***m*** is much > than ***n***
    - actual time complexity is O(m)
- Can we do better?

Radix Sort

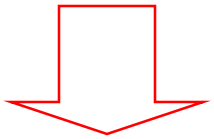
- Suppose:
  - repeatedly sort by digit
  - perform multiple bucket sorts on integers starting with the rightmost digit
  - If maximum value is 999999, only ten buckets (not 1 million) will be necessary
- Use this strategy when the keys are integers, and there is a reasonable limit on their values
  - Number of passes (bucket sort stages) will depend on the number of digits in the maximum value

Example: first pass

12	58	37	64	52	36	99	63	18	9	20	88	47



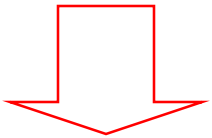
20		12 52	63	64		36	37 47	58 18 88	99 9			

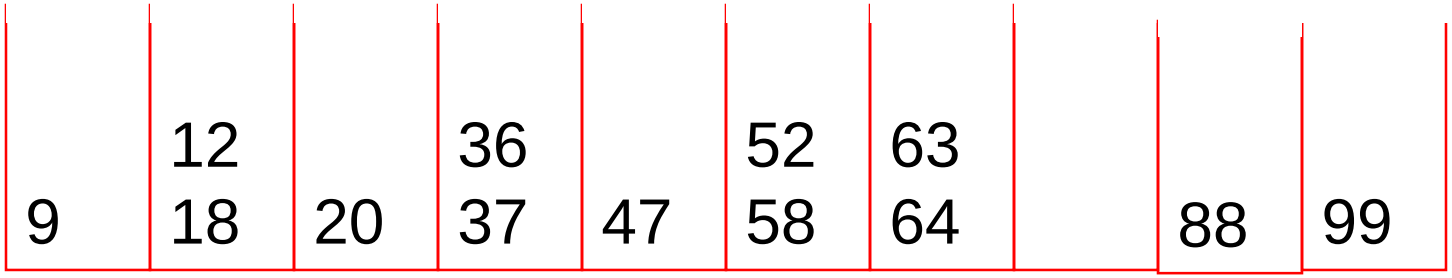


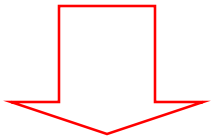
20	12	52	63	64	36	37	47	58	18	88	99	9

Example: second pass

20	12	52	63	64	36	37	47	58	18	88	9	99







9	12	18	20	36	37	47	52	58	63	64	88	99

Radix Sort and Stability

- Radix sort works as long as the bucket sort stages are **stable** sorts
  - Stability is a property of sorts:
  - A sort is stable if it guarantees the relative order of equal items stays the same
  - Given these numbers:

    $[7_1, 6, 7_2, 5, 1, 2, 7_3, -5]$ (subscripts added for clarity)

  - A stable sort would be:

    $[-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]$

Time Complexity

- Given a fixed number of buckets and ***p*** sort stages (number of digits in maximum value), then radix sort is O(n)
  - Each of the ***p*** bucket sort stages take O(n) time
- Strictly speaking, time complexity is O(pn), but ***p*** should be relatively small compared to ***n***
- Note: $p = log_{10}m$, where m is the maximum value in the list

About Radix Sort

- Since the buckets are reused at each stage, only 10 buckets are needed regardless of number of stages
- Radix sort can apply to words
  - Set a limit to the number of letters in a word
  - Use 27 buckets (or more, depending on the letters/characters allowed)
    - one for each letter plus a “blank” character
  - The word-length limit is exactly the number of bucket sort stages needed

Radix Sort Summary

- Bucket sort and Radix sort are O(n) algorithms only because we have imposed restrictions on the input list to be sorted
- Sorting, in general, takes O(n log(n)) time

Final Comments

- Language libraries have sorting algorithms
  - Java Arrays and Collections classes
  - C++ Standard Template Library
- Hybrid sorts
  - When size of unsorted list or portion of array is small:
    - use insertion or selection sort
  - Otherwise:
    - use O(n log(n)) sort like Quicksort of Merge sort
- Many other special case sorting algorithms exist, like radix sort.

Comparison of Various Sorts

Num Items	Selection	Insertion	Quicksort
1000	16	5	0
2000	59	49	6
4000	271	175	5
8000	1056	686	0
16000	4203	2754	11
32000	16852	11039	45
64000	expected?	expected?	68
128000	expected?	expected?	158
256000	expected?	expected?	335
512000	expected?	expected?	722
1024000	expected?	expected?	1550

times in milliseconds

Binary Search (Iterative)

2	3	5	7	11	13	17	19	23	29	31	37	41	43	47	53

```
public static int b-search(int list[], int target)
{       int result = -1;
        int low = 0;
        int high = list.length - 1;
        int mid;
        while( result == -1 && low <= high )
        {       mid = low + ((high - low) / 2);
                if( list[mid] == target )
                        result = mid;
                else if( list[mid] < target)
                        low = mid + 1;
                else
                        high = mid - 1;
        }
        return result;
}
```

Attendance Question 3

Is selection sort always stable?

A. Yes

B. No

Attendance Question 4

- Is the version of insertion sort shown always stable?

A. Yes

B. No

ShellSort

- Created by Donald Shell in 1959
- Wanted to stop moving data small distances (in the case of insertion sort and bubble sort) and stop making swaps that are not helpful (in the case of selection sort)
- Start with sub arrays created by looking at data that is far apart and then reduce the gap size

ShellSort in practice

46 2 83 41 102 5 17 31 64 49 18

Gap of five. Sort sub array with 46, 5, and 18
5 2 83 41 102 18 17 31 64 49 46

Gap still five. Sort sub array with 2 and 17
5 2 83 41 102 18 17 31 64 49 46

Gap still five. Sort sub array with 83 and 31
5 2 31 41 102 18 17 83 64 49 46

Gap still five Sort sub array with 41 and 64
5 2 31 41 102 18 17 83 64 49 46

Gap still five. Sort sub array with 102 and 49
5 2 31 41 49 18 17 83 64 102 46

Continued on next slide:

Completed Shellsort

5 2 31 41 49 18 17 83 64 102 46

Gap now 2: Sort sub array with 5 31 49 17 64 46
5 2 17 41 31 18 46 83 49 102 64

Gap still 2: Sort sub array with 2 41 18 83 102
5 2 17 18 31 41 46 83 49 102 64

Gap of 1 (Insertion sort)

2 5 17 18 31 41 46 49 64 83 102

Array sorted

Shellsort on Another Data Set

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
44	68	191	119	119	37	83	82	191	45	158	130	76	153	39	25

Initial gap = length / 2 = 16 / 2 = 8

initial sub arrays indices:

{0, 8}, {1, 9}, {2, 10}, {3, 11}, {4, 12}, {5, 13}, {6, 14}, {7, 15}
next gap = 8 / 2 = 4

{0, 4, 8, 12}, {1, 5, 9, 13}, {2, 6, 10, 14}, {3, 7, 11, 15}
next gap = 4 / 2 = 2

{0, 2, 4, 6, 8, 10, 12, 14}, {1, 3, 5, 7, 9, 11, 13, 15}

final gap = 2 / 2 = 1

ShellSort Code

```
public static void shellsort(Comparable[] list)
{  Comparable temp; boolean swap;
   for(int gap = list.length / 2; gap > 0; gap /= 2)
      for(int i = gap; i < list.length; i++)
      {   Comparable tmp = list[i];
          int j = i;
          for( ; j >= gap &&
                 tmp.compareTo( list[j - gap] ) < 0;
                 j -= gap )
             list[ j ] = list[ j - gap ];
          list[ j ] = tmp;
       }
}
```

Bucket sort algorithm

Algorithm BucketSort(S)
(values in S are between 0 and m-1)

```
for j ← 0 to m-1 do   // initialize m buckets
    b[j] ← 0
for i ← 0 to n-1 do              // place elements in their
    b[S[i]] ← b[S[i]]  + 1       // appropriate buckets
i ← 0
for j ← 0 to m-1 do   // place elements in buckets
    for r ← 1 to b[j] do         // back in S
          S[i] ← j
          i ← i + 1
```

Bucket sort algorithm

Algorithm BucketSort(S)
(S is an array of entries whose keys are between 0..m-1)

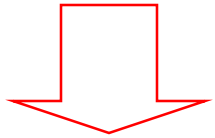
```
for j ← 0 to m-1 do              // initialize m buckets
    initialize queue b[j]
for i ← 0 to n-1 do                        // place in buckets
    b[S[i].getKey()].enqueue( S[i] );
i ← 0
for j ← 0 to m-1 do              // place elements in
    while not b[j].isEmpty() do          // buckets back in S
            S[i] ← b[j].dequeue()
            i ← i + 1
```

Stable Sorting

- A property of sorts
- If a sort guarantees the relative order of equal items stays the same then it is a *stable sort*
- [7_1, 6, 7_2, 5, 1, 2, 7_3, -5]
  - subscripts added for clarity
- [-5, 1, 2, 5, 6, 7_1, 7_2, 7_3]
  - result of stable sort
- Real world example:
  - sort a table in Wikipedia by one criteria, then another
  - sort by country, then by major wins

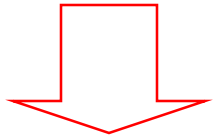
Example: 1st and 2nd passes

12	58	37	64	52	36	99	63	18	9	20	88	47



sort by rightmost digit

20	12	52	63	64	36	37	47	58	18	88	9	99



sort by leftmost digit

9	12	18	20	36	37	47	52	58	63	64	88	99