# Analysis of Algorithms

CS 1037a – Topic 13

# Overview

- Time complexity
  - exact count of operations T(n) as a function of input size n
  - complexity analysis using O(...) bounds
  - constant time, linear, logarithmic, exponential,… complexities
- Complexity analysis of basic data structures' operations
- *Linear* and *Binary Search* algorithms and their analysis
- Basic Sorting algorithms and their analysis

# Related materials

from Main and Savitch

*“Data Structures & other objects using C++”*

- Sec. 12.1: Linear (serial) search, Binary search
- Sec. 13.1: Selection and Insertion Sort

# Analysis of Algorithms

- *Efficiency* of an algorithm can be measured in terms of:
  - Execution time (*time complexity*)
  - The amount of memory required (*space complexity*)
- Which measure is more important?
  - Answer often depends on the limitations of the technology available at time of analysis

# Time Complexity

- For most of the algorithms associated with this course, time complexity comparisons are more interesting than space complexity comparisons
- *Time complexity*: A measure of the amount of time required to execute an *algorithm*

# Time Complexity

- Factors that *should not* affect time complexity analysis:
  - The programming language chosen to implement the algorithm
  - The quality of the compiler
  - The speed of the computer on which the algorithm is to be executed

# Time Complexity

- Time complexity analysis for an algorithm is *independent* of programming language,machine used
- *Objectives* of time complexity analysis:
  - To determine the feasibility of an algorithm by estimating an *upper bound* on the amount of work performed
  - To compare different algorithms before deciding on which one to implement

# Time Complexity

- Analysis is based on the amount of *work* done by the algorithm
- Time complexity expresses the relationship between the *size of the input* and the *run time* for the algorithm
- Usually expressed as a proportionality, rather than an exact function

# Time Complexity

- To simplify analysis, we sometimes ignore work that takes a *constant* amount of time, independent of the problem input size
- When comparing two algorithms that perform the same task, we often just concentrate on the *differences* between algorithms

# Time Complexity

- Simplified analysis can be based on:
  - Number of arithmetic operations performed
  - Number of comparisons made
  - Number of times through a critical loop
  - Number of array elements accessed
  - etc

# Example: Polynomial Evaluation

Suppose that exponentiation is carried out using multiplications. Two ways to evaluate the polynomial

$p(x) = 4x^4 + 7x^3 – 2x^2 + 3x^1 + 6$

are:

*Brute force method*:

p(x) = 4*x*x*x*x + 7*x*x*x – 2*x*x + 3*x + 6

*Horner's method*:

p(x) = (((4*x + 7) * x – 2) * x + 3) * x + 6

# Example: Polynomial Evaluation

Method of analysis:

- Basic operations are multiplication, addition, and subtraction
- We'll only consider the number of multiplications, since the number of additions and subtractions are the same in each solution
- We'll examine the general form of a polynomial of degree *n*, and express our result in terms of *n*
- We'll look at the *worst case* (max number of multiplications) to get an *upper bound* on the work

# Example: Polynomial Evaluation

*General form* of polynomial is

$$p(x) = a_n x^n + a_{n-1} x^{n-1} + a_{n-2} x^{n-2} + \ldots + a_1 x^1 + a_0$$

where $a_n$ is non-zero for all n >= 0

# Example: Polynomial Evaluation

Analysis for *Brute Force Method*:

| | |
|---|---|
| $p(x) = a_n \underline{* x * x * \ldots * x * x} +$ | n multiplications |
| $a_{n-1} \underline{* x * x * \ldots * x * x} +$ | n-1 multiplications |
| $a_{n-2} \underline{* x * x * \ldots * x * x} +$ | n-2 multiplications |
| … + | … |
| $a_2 \underline{* x * x} +$ | 2 multiplications |
| $a_1 \underline{* x} +$ | 1 multiplication |
| $a_0$ | |

# Example: Polynomial Evaluation

Number of multiplications needed in the worst case is

$T(n) = n + n-1 + n-2 + \ldots + 3 + 2 + 1$

$= n(n + 1)/2$ (*result from high school math ***)

$= n^2/2 + n/2$

This is an exact formula for the maximum number of multiplications. In general though, analyses yield upper bounds rather than exact formulae. We say that the number of multiplications is *on the order of $n^2$*, or *$O(n^2)$*. (Think of this as being *proportional to* $n^2$.)

*(** We'll give a proof for this result a bit later)*

# Example: Polynomial Evaluation

Analysis for *Horner’s Method*:

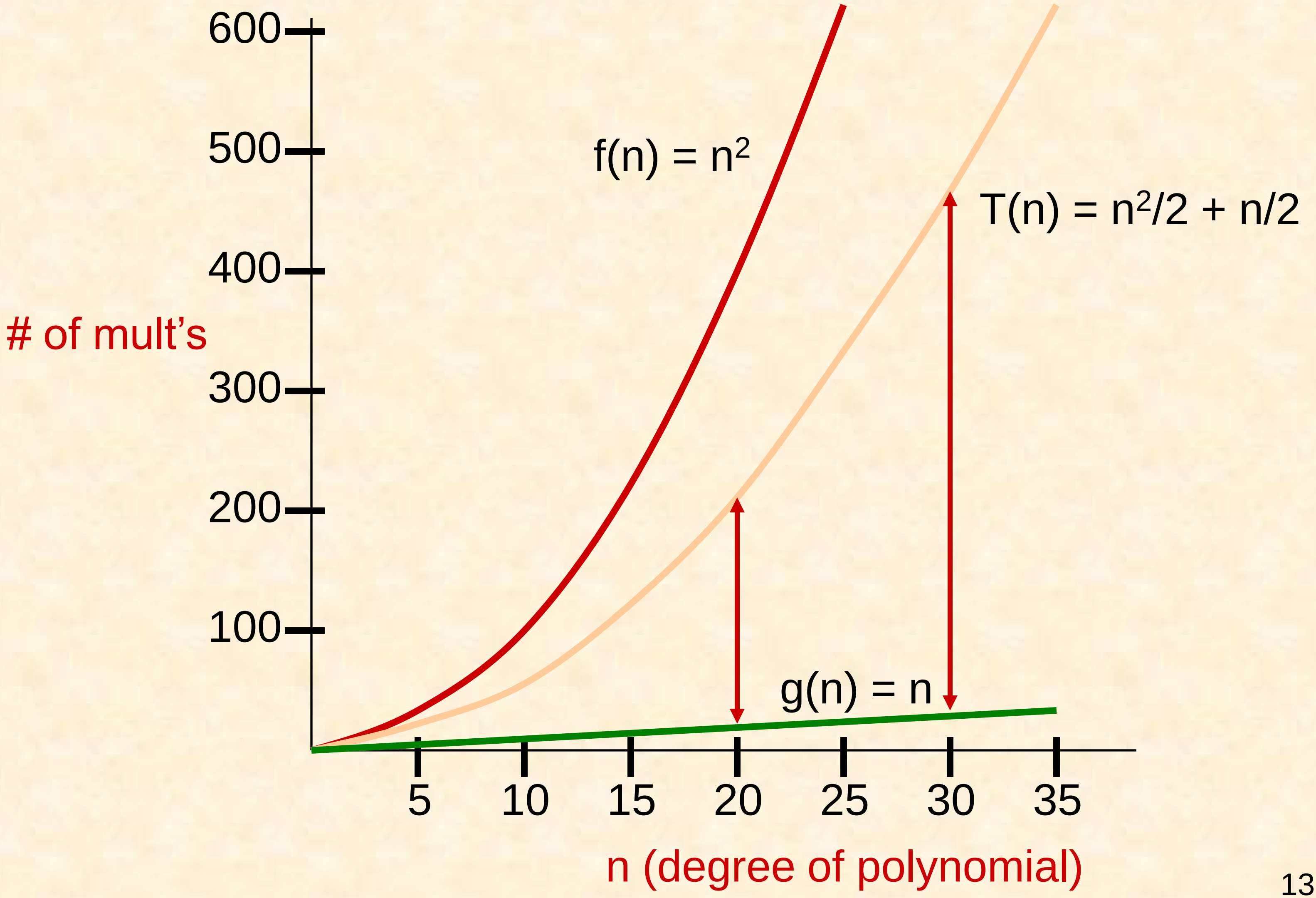
| | | |
|---|---|---|
| $p(x) = ( \ldots ((( a_n * x +$ | 1 multiplication | |
| $a_{n-1}) * x +$ | 1 multiplication | |
| $a_{n-2}) * x +$ | 1 multiplication | |
| $\ldots +$ | | *n times* |
| $a_2) * x +$ | 1 multiplication | |
| $a_1) * x +$ | 1 multiplication | |
| $a_0$ | | |

*T(n) = n*, so the number of multiplications is *O(n)*

# Example: Polynomial Evaluation

| n<br>(Horner) | $n^2/2 + n/2$<br>(brute force) | $n^2$ |
|---|---|---|
| 5 | 15 | 25 |
| 10 | 55 | 100 |
| 20 | 210 | 400 |
| 100 | 5050 | 10000 |
| 1000 | 500500 | 1000000 |

# Example: Polynomial Evaluation



# Sum of First **n** Natural Numbers

Write down the terms of the sum in forward and reverse orders; there are n terms:

$$\begin{array}{ccccccccccccc} T(n) & = & 1 & + & 2 & + & 3 & + \ldots + & (n-2) & + & (n-1) & + & n \\ T(n) & = & n & + & (n-1) & + & (n-2) & + \ldots + & 3 & + & 2 & + & 1 \end{array}$$

Add the terms in the boxes to get:

$2*T(n) = (n+1) + (n+1) + (n+1) + \ldots + (n+1) + (n+1) + (n+1)$

$= n(n+1)$

Therefore, $T(n) = (n*(n+1))/2 = n^2/2 + n/2$

# Big-O Notation

- Formally, the *time complexity* T(n) of an algorithm is *O(f(n))* (***of the order f(n)***) if, for some positive constants $C_1$ and $C_2$ for all but finitely many values of n

$$C_1*f(n) \leq T(n) \leq C_2*f(n)$$

- This gives *upper* and *lower bounds* on the amount of work done for all sufficiently large n

# Big-O Notation

***Example***: Brute force method for polynomial evaluation: We chose the highest-order term of the expression $T(n) = n^2/2 + n/2$, with a coefficient of $1$, so that $f(n) = n^2$.

$T(n)/n^2$ approaches $1/2$ for large $n$, so $T(n)$ is approximately $n^2/2$.

$$n^2/2 <= T(n) <= n^2$$

so $T(n)$ is $O(n^2)$.

# Big-O Notation

- We want an easily recognized elementary function to describe the performance of the algorithm, so we use the *dominant term* of T(n): it determines the basic *shape* of the function

# Worst Case vs. Average Case

- *Worst case analysis* is used to find an upper bound on algorithm performance for large problems (large n)
- *Average case analysis* determines the average (or *expected*) performance
- Worst case time complexity is usually simpler to work out

# Big-O Analysis in General

- With *independent* nested loops: The number of iterations of the inner loop is independent of the number of iterations of the outer loop
- *Example*:

```
int x = 0;

for ( int j = 1; j <= n/2; j++ )

    for ( int k = 1; k <= n*n; k++ )

        x = x + j + k;
```

Outer loop executes n/2 times. For each of those times, inner loop executes $n^2$ times, so the body of the inner loop is executed $(n/2)*n^2 = n^3/2$ times. The algorithm is $O(n^3)$ .

# Big-O Analysis in General

- With *dependent* nested loops: Number of iterations of the inner loop depends on a value from the outer loop
- *Example*:

```
int x = 0;
for ( int j = 1; j <= n; j++ )
    for ( int k = 1; k < 3*j; k++ )
        x = x + j;
```

When j is 1, inner loop executes 3 times; when j is 2, inner loop executes 3*2 times; … when j is n, inner loop executes 3*n times. In all the inner loop executes 3+6+9+…+3n = 3(1+2+3+…+n) = $3n^2/2 + 3n/2$ times. The algorithm is $O(n^2)$.

# Big-O Analysis in General

Assume that a computer executes a million instructions a second. This chart summarizes the amount of time required to execute f(n) instructions on this machine for various values of n.

| f(n) | $n=10^3$ | $n=10^5$ | $n=10^6$ |
|---|---|---|---|
| $\log_2(n)$ | $10^{-5}$ sec | $1.7 * 10^{-5}$ sec | $2 * 10^{-5}$ sec |
| n | $10^{-3}$ sec | 0.1 sec | 1 sec |
| $n*\log_2(n)$ | 0.01 sec | 1.7 sec | 20 sec |
| $n^2$ | 1 sec | 3 hr | 12 days |
| $n^3$ | 17 min | 32 yr | 317 centuries |
| $2^n$ | $10^{285}$ centuries | $10^{10000}$ years | $10^{100000}$ years |

# Big-O Analysis in General

- To determine the time complexity of an algorithm:
  - Express the amount of work done as a sum $f_1(n) + f_2(n) + \ldots + f_k(n)$
  - Identify the *dominant term*: the $f_i$ such that $f_j$ is $O(f_i)$ and for $k$ different from $j$

    $f_k (n) < f_j (n)$ (for all sufficiently large n)
  - Then the time complexity is $O(f_i)$

# Big-O Analysis in General

- Examples of *dominant terms*:

n dominates $\log_2(n)$

$n*\log_2(n)$ dominates n

$n^2$ dominates $n*\log_2(n)$

$n^m$ dominates $n^k$ when m > k

$a^n$ dominates $n^m$ for any a > 1 and m >= 0

- That is, $\log_2(n) < n < n*\log_2(n) < n^2 < \ldots < n^m < a^n$ for a >= 1 and m > 2

# Intractable problems

- A problem is said to be *intractable* if solving it by computer is impractical
- ***Example***: Algorithms with time complexity $O(2^n)$ take too long to solve even for moderate values of $n$; a machine that executes 100 million instructions per second can execute $2^{60}$ instructions in about 365 years

# Constant Time Complexity

- Algorithms whose solutions are independent of the size of the problem's inputs are said to have *constant* time complexity

- Constant time complexity is denoted as O(1)

# Time Complexities for Data Structure Operations

- Many operations on the data structures we’ve seen so far are clearly O(1): retrieving the size, testing emptiness, etc

- We can often recognize the time complexity of an operation that modifies the data structure without a formal proof

# Time Complexities for Array Operations

- Array elements are stored contiguously in memory, so the time required to compute the memory address of an array element arr[k] is independent of the array's size: It's the *start address* of arr plus k * (size of an individual element)
- So, storing and retrieving array elements are O(1) operations

# Time Complexities for Array-Based List Operations

- Assume an n-element List (Topic 8):
  - *insert* operation is O(n) in the worst case, which is adding to the first location: all n elements in the array have to be shifted one place to the right before the new element can be added

# Time Complexities for Array-Based List Operations

- Inserting into a full List is also O(n):
  - replaceContainer copies array contents from the old array to a new one (O(n))
  - All other activies (allocating the new array, deleting the old one, etc) are O(1)
  - Replacing the array and then inserting at the beginning requires O(n) + O(n) time, which is O(n)

# Time Complexities for Array-Based List Operations

- *remove* operation is O(n) in the worst case, which is removing from the first location: n-1 array elements must be shifted one place left
- *retrieve*, *replace*, and *swap* operations are O(1): array indexing allows direct access to an array location, independent of the array size; no shifting occurs
- *find* is O(n) because the entire list has to be searched in the worst case

# Time Complexities for Linked List Operations

- Singly linked list with n nodes:
  - *addHead*, *removeHead*, and *retrieveHead* are all O(1)
  - *addTail* and *retrieveTail* are O(1) ***provided that*** the implementation has a tail reference; otherwise, they're O(n)
  - *removeTail* is O(n): need to traverse to the second-last node so that its reference can be reset to *NULL*

# Time Complexities for Linked List Operations

- Singly linked list with n nodes (cont'd):
    - Operations to access an item by position (*add* , *retrieve*, *remove(unsigned int k)*, *replace*) are O(n): need to traverse the whole list in the worst case
    - Operations to access an item by its value (*find*, *remove(Item item)*) are O(n) for the same reason

# Time Complexities for Linked List Operations

- Doubly-linked list with n nodes:
  - Same as for singly-linked lists, except that ***all*** head and tail operations, including *removeTail*, are O(1)
- Ordered linked list with n nodes:
  - Comparable operations to those found in the linked list class have the same time complexities
  - *add(Item item)* operation is O(n): may have to traverse the whole list

# Time Complexities for Bag Operations

- Assume the bag contains n items, then
- *add*:
  - O(1) for our array-based implementation: new item is added to the end of the array
  - If the bag can grow arbitrarily large (*i.e.:* if we replace the underlying array), adding to a “full” bag is O(n)
  - Also O(1) if we add to end of an array-based list, or head or tail of a linked list

# Time Complexities for Bag Operations

- *getOne*:
  - Must be careful to ensure that it is O(1) if we use an underlying array or array-based list
  - Don’t shift array or list contents
  - Retrieve the $k^{th}$ item, copy the $n^{th}$ item into the $k^{th}$ position, and remove the $n^{th}$ item
  - Worst case is O(n) for any linked list implementation: requires list traversal

# Time Complexities for Bag Operations

- Copy constructor for Bags (Topic 4, slide 4-33):
  - Algorithm is O(n) where n is the number of items copied
  - But, copying the underlying items may not be an O(1) task: it depends on the kind of item being copied
  - For class Bag<Item>: if copying an underlying item is O(m), then the time complexity for the copy constructor is O(n*m)

# Time Complexities for Stack Operations

- Stack using an underlying array:
  - All operations are O(1), provided that the top of the stack is always at the highest index currently in use: no shifting required
- Stack using an array-based list:
  - All operations O(1), provided that the tail of the list is the top of the stack
  - ***Exception***: *push* is O(n) if the array size has to double

# Time Complexities for Stack Operations

- Stack using an underlying linked list:
  - All operations are, or should be, O(1)
  - Top of stack is the head of the linked list
  - If a doubly-linked list with a tail pointer is used, the top of the stack can be the tail of the list

# Time Complexities for Queue Operations

- Queue using an underlying array-based list:
  - *peek* is O(1)
  - *enqueue* is O(1) unless the array size has to be increased (in which case it's O(n))
  - *dequeue* is O(n) : all the remaining elements have to be shifted

# Time Complexities for Queue Operations

- Queue using an underlying linked list:
  - As long as we have both a head and a tail pointer in the linked list, all operations are O(1)
    - important: *enqueue()* should use *addTail()*
      *dequeue()* should use *removeHead()*

      Why not the other way around?
  - No need for the list to be doubly-linked

# Time Complexities for Queue Operations

- Circular queue using an underlying array:
  - All operations are O(1)
  - If we revise the code so that the queue can be arbitrarily large, enqueue is O(n) on those occasions when the underlying array has to be replaced

# Time Complexities for OrderedList Operations

OrderedList with array-based m_container:

- Our implementation of *insert(item)* (see slide 10-12) uses “***linear search***” that traverses the list from its beginning until the right spot for the new item is found – linear complexity O(n)
- Operation *remove(int pos)* is also O(n) since items have to be shifted in the array

# Basic Search Algorithms and their Complexity Analysis

# Linear Search: Example 1

- *The problem*: Search an array a of size n to determine whether the array contains the value key; return index if found, -1 if not found

```
Set k to 0.
While (k < n) and (a[k] is not key)
   Add 1 to k.
If  k == n  Return –1.
Return k.
```

# Linear Search: Example 2
# “find” in Array Based List

```
template <class Item>  template <class Equality>
int List<Item>::find(Item key) const   {
      for (int k = 1; k<= getLength(); i++)
               if ( Equality::compare(m_container[k], key)  )   return k;
      return –1;
}
```

// this extra function requires additional templated
// argument for a comparison functor whose method
// *compare* checks two items for equality (as in slide 11-79)

# Example of using *LinearSearch*

```
int main()   {
        List<int> mylist;
        … // code adding some ints into mylist
        cout << mylist.find<IsEqual>(5);
}
```

Additional templated argument for function *find()* should be specified in your code

# Analysis of Linear Search

- Total amount of work done:
  - Before loop: a constant amount a
  - Each time through loop: 2 comparisons, an and operation, and an addition: a constant amount of work b
  - After loop: a constant amount c
  - In worst case, we examine all n array locations, so T(n) = a +b*n + c = b*n + d, where d = a+c, and time complexity is O(n)

# Analysis of Linear Search

- Simpler (less formal) analysis:
  - Note that work done before and after loop is independent of n, and work done during a single execution of loop is independent of n
  - In worst case, loop will be executed n times, so amount of work done is proportional to n, and algorithm is O(n)

# Analysis of Linear Search

- *Average* case for a *successful* search:
  - Probability of key being found at index k is 1 in n for each value of k
  - Add up the amount of work done in each case, and divide by total number of cases:

((a*1+d) + (a*2+d) + (a*3+d) + … + (a*n+d))/n

= (n*d + a*(1+2+3+ … +n))/n

= n*d/n + a*(n*(n+1)/2)/n = d + a*n/2 + a/2 = (a/2)*n + e, where constant e=d+a/2, so expected case is also O(n)

# Analysis of Linear Search

- Simpler approach to expected case:
  - Add up the number of times the loop is executed in each of the n cases, and divide by the number of cases n
  - (1+2+3+ … +(n-1)+n)/n = (n*(n+1)/2)/n = n/2 + 1/2; algorithm is therefore O(n)

# Linear Search for LinkedList

- Linear search can be also done for *LinkedList*
  - **exercise**: write code for function

```
template <class Item>  template <class Equality>
int LinkedList<Item>::find(Item key) const   { …}
```

- Complexity of function *find(key)* for class *LinkedList* should also be O(n)

# Binary Search
## (on sorted arrays)

- General case: search a sorted array a of size n looking for the value key
- *Divide and conquer* approach:
  - Compute the middle index mid of the array
  - If key is found at mid, we're done
  - Otherwise repeat the approach on the half of the array that might still contain key
  - etc…

# Example: Binary Search For Ordered List

// A new member function for class OrderedList<Item,Order>

```
template <class Item, class Order>
int OrderedList<Item,Order>::binarySearch(Item key)  const   {
      int first = 1,    last = m_container.getLength();
      while (first <= last) {     // start of while loop
           int mid    =  (first+last)/2;
           Item val  =  retrieve(mid);
           if          ( Order::compare(key ,  val) )     last = mid-1;
           else if ( Order::compare(val  ,  key) )    first = mid+1;
           else                                         return  mid;
     }      // end of while loop
     return –1;
}
```

# Analysis of Binary Search

- The amount of work done before and after the loop is a constant, and independent of n
- The amount of work done during a single execution of the loop is constant
- Time complexity will therefore be proportional to number of times the loop is executed, so that’s what we’ll analyze

# Analysis of Binary Search

- *Worst case*: key is not found in the array
- Each time through the loop, at least half of the remaining locations are rejected:
  - After first time through, <= n/2 remain
  - After second time through, <= n/4 remain
  - After third time through, <= n/8 remain
  - After k^th time through, <= $n/2^k$ remain

# Analysis of Binary Search

- Suppose in the worst case that maximum number of times through the loop is $k$; we must express $k$ in terms of $n$
- Exit the do..while loop when number of remaining possible locations is less than 1 (that is, when first > last): this means that $n/2^k < 1$

# Analysis of Binary Search

- Also, $n/2^{k-1} >= 1$; otherwise, looping would have stopped after $k-1$ iterations
- Combining the two inequalities, we get:

$$n/2^k < 1 <= n/2^{k-1}$$

- Invert and multiply through by $n$ to get:

$$2^k > n >= 2^{k-1}$$

# Analysis of Binary Search

- Next, take base-2 logarithms to get:

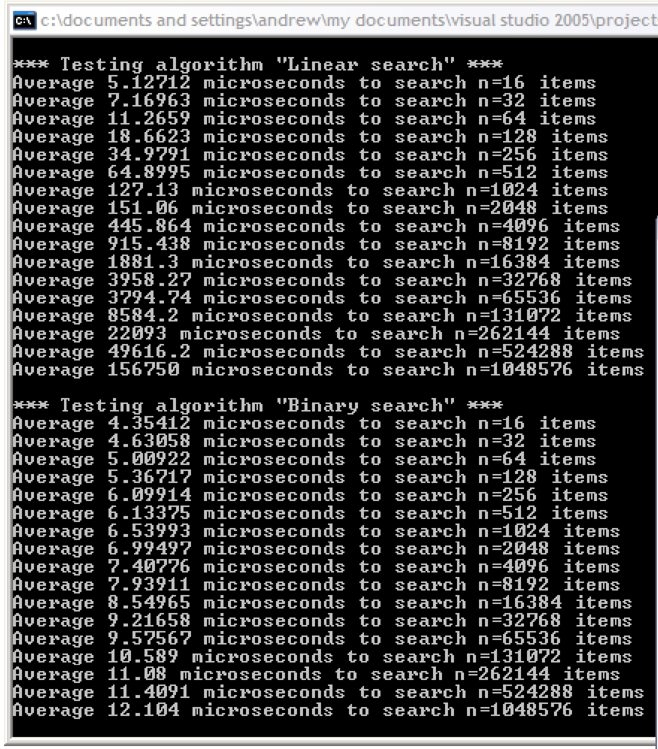
  $k > \log_2(n) >= k-1$

- Which is equivalent to:

  $\log_2(n) < k <= \log_2(n) + 1$

- Thus, binary search algorithm is $O(\log_2(n))$ in terms of the number of array locations examined

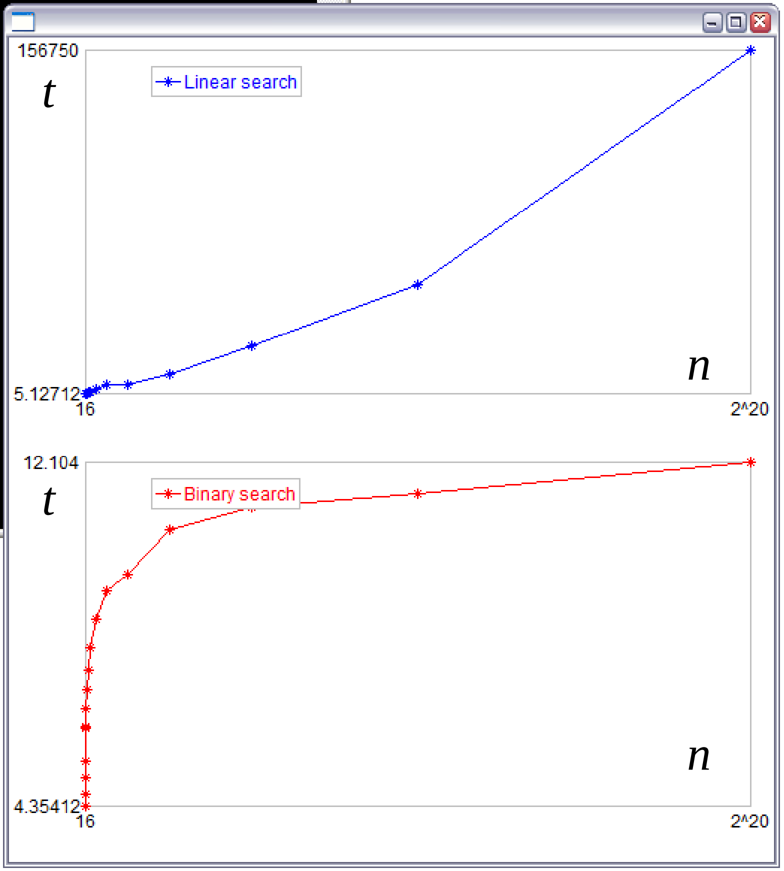
# Binary vs. Liner Search



c:\documents and settings\andrew\my documents\visual studio 2005\projects\topic13_analysisofalgorithm...

```
*** Testing algorithm "Linear search" ***
Average 5.12712 microseconds to search n=16 items
Average 7.16963 microseconds to search n=32 items
Average 11.2659 microseconds to search n=64 items
Average 18.6623 microseconds to search n=128 items
Average 34.9791 microseconds to search n=256 items
Average 64.8995 microseconds to search n=512 items
Average 127.13 microseconds to search n=1024 items
Average 151.06 microseconds to search n=2048 items
Average 445.864 microseconds to search n=4096 items
Average 915.438 microseconds to search n=8192 items
Average 1881.3 microseconds to search n=16384 items
Average 3958.27 microseconds to search n=32768 items
Average 3794.74 microseconds to search n=65536 items
Average 8584.2 microseconds to search n=131072 items
Average 22093 microseconds to search n=262144 items
Average 49616.2 microseconds to search n=524288 items
Average 156750 microseconds to search n=1048576 items

*** Testing algorithm "Binary search" ***
Average 4.35412 microseconds to search n=16 items
Average 4.63058 microseconds to search n=32 items
Average 5.00922 microseconds to search n=64 items
Average 5.36717 microseconds to search n=128 items
Average 6.09914 microseconds to search n=256 items
Average 6.13375 microseconds to search n=512 items
Average 6.53993 microseconds to search n=1024 items
Average 6.99497 microseconds to search n=2048 items
Average 7.40776 microseconds to search n=4096 items
Average 7.93911 microseconds to search n=8192 items
Average 8.54965 microseconds to search n=16384 items
Average 9.21658 microseconds to search n=32768 items
Average 9.57567 microseconds to search n=65536 items
Average 10.589 microseconds to search n=131072 items
Average 11.08 microseconds to search n=262144 items
Average 11.4091 microseconds to search n=524288 items
Average 12.104 microseconds to search n=1048576 items
```



*search* for one out of *n* ordered integers

**see demo:** *www.csd.uwo.ca/courses/CS1037a/demos.html*

# Improving *insert* in *OrderedList*

- Function *insert( item )* for *OrderedList* (see slide 10-12) can use *binary search* algorithm (instead of *linear search*) when looking for the "right" place for the new item inside *m_container* (an array-based *List)*

**Question**: would worst-case complexity of *insert* improve from O(n) to O($\log_2$(n))?

**Answer**: NO!

we can find the "right" position *k* faster, but *m_container.insert(k,item)* still requires shifting of O(n) items in the underlying array

# Improving *insert* in *OrderedList*

**Question**: would it be possible to improve complexity of *insert* from O(n) to O($\log_2$(n)) if we used *m_container* of class *LinkedList* ?

**Answer**: still NO!

**in this case we cannot even do *Binary Search* efficiently in O($\log_2$(n))**

*- finding an item in the "middle" of the linked list requires linear traversal*

*- in contrast, accessing "middle" item in an array is a one step operation*

*e.g. m_container[k] or *(m_container+k)*

In topic 15 we will study a new data structure for storing ordered items (*BST*) which is better than our *OrderedList*

- operations *insert* and *remove* in *BST* are

  $O(\log_2(n))$

# Basic Sorting Algorithms and their Complexity Analysis

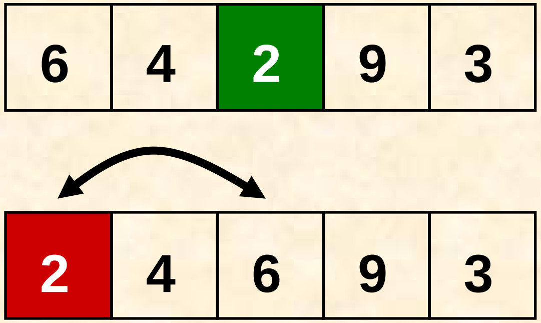
# Analysis: Selection Sort Algorithm

- Assume we have an unsorted collection of *n* elements in an array or list called *container*; elements are either of a simple type, or are pointers to data
- Assume that the elements can be compared in size ( <, >, ==, *etc*)
- Sorting will take place “in place” in *container*

🟥 - sorted portion of the list

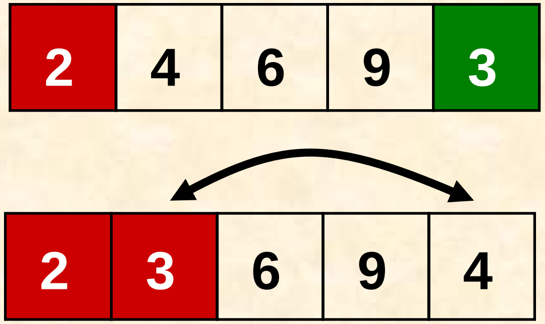
🟩 - minimum element in unsorted portion

# Analysis: Selection Sort Algorithm



Find smallest element in unsorted portion of container

Interchange the smallest element with the one at the front of the unsorted portion



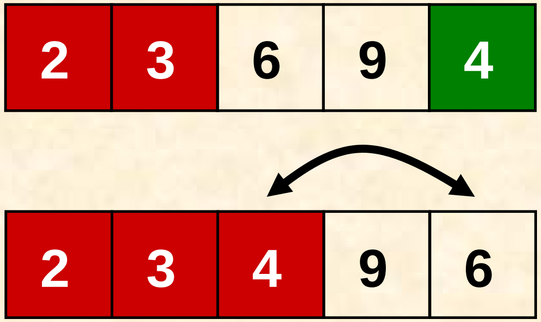
Find smallest element in unsorted portion of container

Interchange the smallest element with the one at the front of the unsorted portion

🟥 - sorted portion of the list

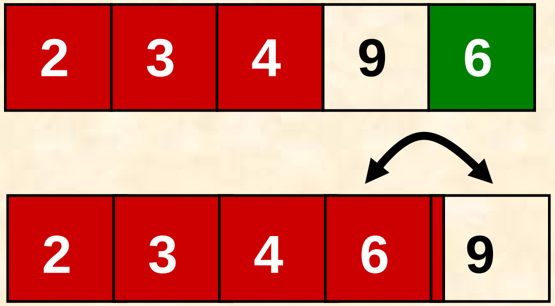
🟩 - minimum element in unsorted portion

# Analysis: Selection Sort Algorithm



Find smallest element in unsorted portion of container

Interchange the smallest element with the one at the front of the unsorted portion



Find smallest element in unsorted portion of container

Interchange the smallest element with the one at the front of the unsorted portion

**After n-1 repetitions of this process, the last item has automatically fallen into place**

# Selection Sort for (array-based) List

// A new member function for class List<Item>, needs additional template parameter

```
template <class Item> template <class Order>
void List<Item>::selectionSort() {
    unsigned int minSoFar, i, k;
    for (i = 1; i < getLength(); i++ )   { // ‘unsorted’ part starts at given ‘i’
        minSoFar = i;
        for (k = i+1; k <= getLength(); k++)    // searching for min Item inside ‘unsorted’
            if (  Order::compare(retrieve(k),retrieve(minSoFar))  )  minSoFar = k;
        swap( i, minSoFar );   // reminder: “swap” switches Items in 2 given positions
     } // end of for-i loop
}
```

see slide 8-22

# Example of applying *selectionSort* to a list

```
int main()   {
        List<int> mylist;
        … // code adding some into list a
        mylist.selectionSort<IsLess>();
}
```

additional templated argument for function *selectionSort()* should be specified in your code

# Analysis: Selection Sort Algorithm

- We'll determine the time complexity for selection sort by counting the number of data items examined in sorting an n-item array or list
- Outer loop is executed n-1 times
- Each time through the outer loop, one more item is sorted into position

# Analysis: Selection Sort Algorithm

- On the k$^{th}$ time through the outer loop:
  - Sorted portion of container holds k-1 items initially, and unsorted portion holds n-k+1
  - Position of the first of these is saved in minSoFar; data object is not examined
  - In the inner loop, the remaining n-k items are compared  to the one at minSoFar to decide if minSoFar has to be reset

# Analysis: Selection Sort Algorithm

  - 2 data objects are examined each time through the inner loop
  - So, in total, 2*(n-k) data objects are examined by the inner loop during the $k^{th}$ pass through the outer loop
- Two elements may be switched following the inner loop, but the data values aren’t examined (compared)

# Analysis: Selection Sort Algorithm

- Overall, on the $k^{th}$ time through the outer loop, 2*(n-k) objects are examined
- But k ranges from 1 to n-1 (the number of times through the outer loop)
- Total number of elements examined is:

**T(n)** = 2*(n-1) + 2*(n-2) + 2*(n-3) + … + 2*(n-(n-2)) + 2*(n-(n-1))

= 2*((n-1) + (n-2) + (n-3) + … + 2 + 1) *(or 2*(sum of first n-1 ints)*

= 2*((n-1)*n)/2) = $\mathbf{n^2 - n}$, so the algorithm is $\mathbf{O(n^2)}$

# Analysis: Selection Sort Algorithm

- This analysis works for both arrays and array-based lists, provided that, in the list implementation, we either directly access array m_container, or use retrieve and replace operations (O(1) operations) rather than insert and remove (O(n) operations)

# Analysis: Selection Sort Algorithm

- The algorithm has deterministic complexity
  - the number of operations does not depend on specific items, it depends only on the number of items
  - all possible instances of the problem (“best case”, “worst case”, “average case”) give the same number of operations **$T(n)=n^2–n=O(n^2)$**

# Insertion Sort Algorithm

- items are sorted on "insertion", for example

```
List<int> a;

    ……………

OrderedList<int> sorted;

while (!a.isEmpty()) sorted.insert( a.popBack() ); // sorting on insertion

while (!sorted.isEmpty()) a.append( sorted.popBack() );
```

O(n) complexity operation “insert”
performed n times inside “while-loop”
=> overall complexity is $O(n^2)$

In this case, content of list *a* is sorted using one extra container (not *"in place"* sorting)

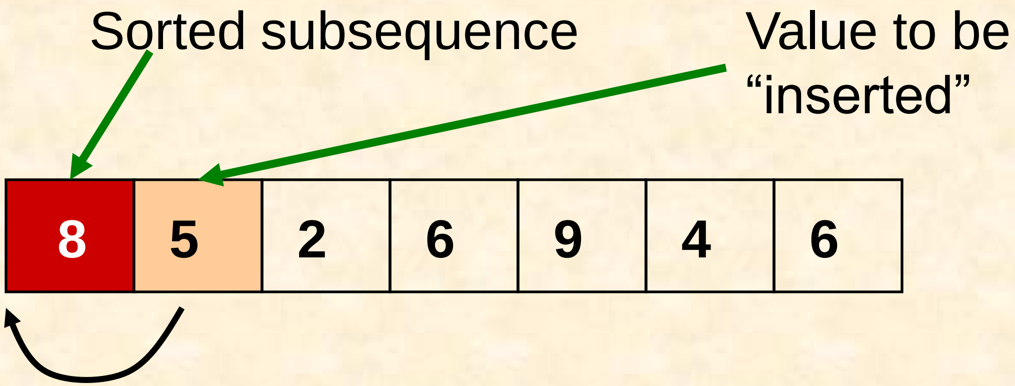
# Insertion Sort Algorithm

- Same approach can be also implemented *in-place* using existing container that is not in order:
  - Front item in sequence is a *sorted subsequence* of length 1
  - Second item of sequence is “inserted” into the sorted subsequence, which is now of length 2
  - Process repeats, always inserting the first item from the unsorted portion into the sorted subsequence, until the entire sequence is in order

🟥 - sorted portion of the list

🟧 - first element in unsorted portion of the list

# Insertion Sort Algorithm



Sorted subsequence

Value to be "inserted"

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 8 | 5 | 2 | 6 | 9 | 4 | 6 |

Again, we're sorting in ascending order of int

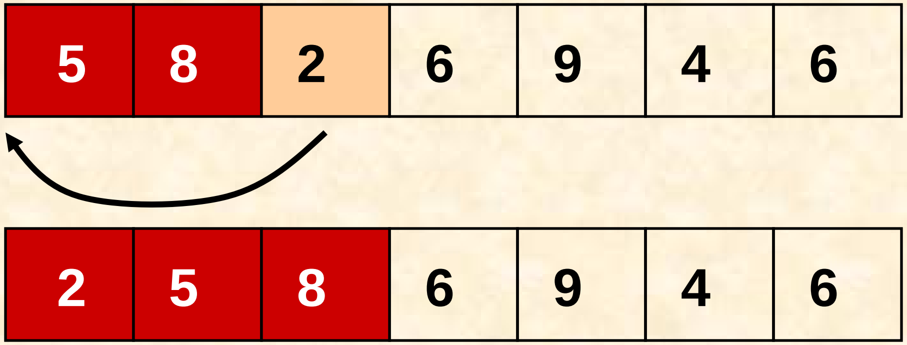
Value 5 is to be inserted where the 8 is; reference to 8 will be copied to where the 5 is, the 5 will be put in the vacated position, and the sorted subsequence now has length 2

|   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|
| 5 | 8 | 2 | 6 | 9 | 4 | 6 |

🟥 - sorted portion of the list

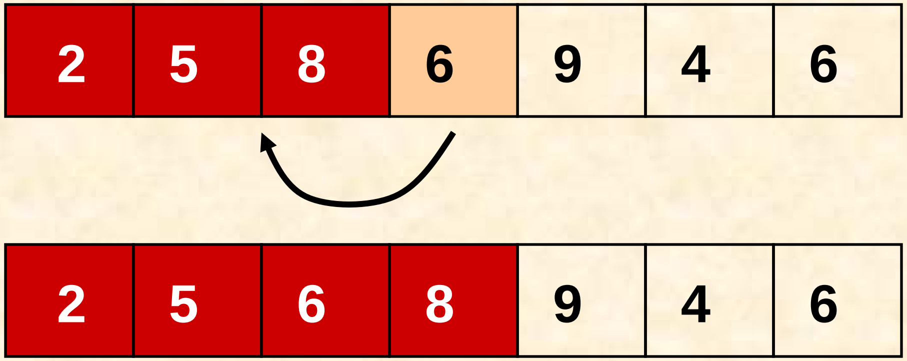
🟧 - first element in unsorted portion of the list

# Insertion Sort Algorithm



| | | | | | | |
|---|---|---|---|---|---|---|
| 5 | 8 | 2 | 6 | 9 | 4 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 5 | 8 | 6 | 9 | 4 | 6 |



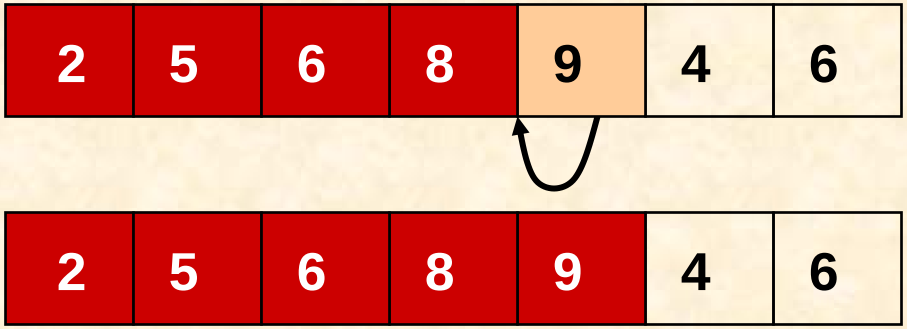
| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 5 | 8 | 6 | 9 | 4 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 5 | 6 | 8 | 9 | 4 | 6 |

🟥 - sorted portion of the list

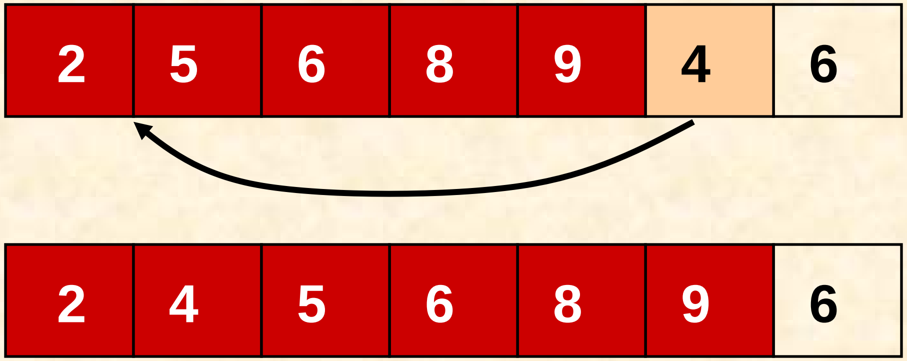
🟧 - first element in unsorted portion of the list

# Insertion Sort Algorithm



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 5 | 6 | 8 | 9 | 4 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 5 | 6 | 8 | 9 | 4 | 6 |



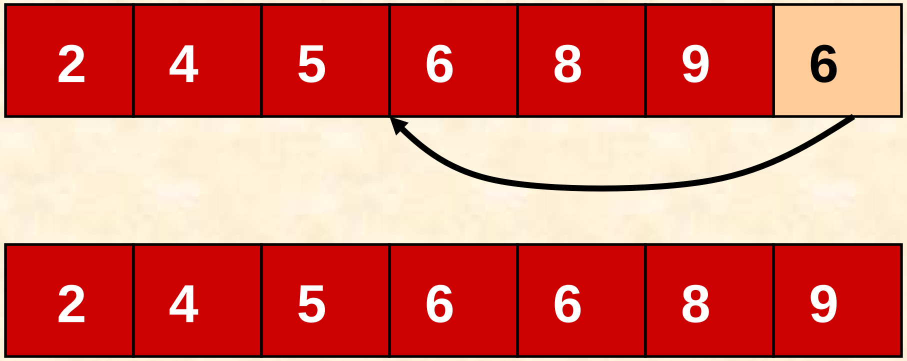
| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 5 | 6 | 8 | 9 | 4 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 5 | 6 | 8 | 9 | 6 |

🟥 -  sorted portion of the list

🟧 - first element in unsorted portion of the list

# Insertion Sort Algorithm



| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 5 | 6 | 8 | 9 | 6 |

| | | | | | | |
|---|---|---|---|---|---|---|
| 2 | 4 | 5 | 6 | 6 | 8 | 9 |

**We’re done !**

# In-place Insertion Sort For Array-Based List

// A new member function for class List<Item>, needs additional template parameter

```
template <class Item> template <class Order>
void List<Item>::insertionSort() {
    unsigned int i, k;
    for (i = 2; i <= getLength(); i++ )   {  // item 'i' will move into 'sorted'
        for ( k = i-1; k >0; k--)  {
            if ( Order::compare(  retrieve(k),  retrieve(k+1)  ))    break;
            else  swap(k,k+1);     // shifting  i-th item  "down"  until  the
        }                          // "right" spot in 'sorted'  1 <= k <= (i-1)
    }
}
```

# Analysis: Insertion Sort Algorithm

- the worst case complexity of *insertionSort()* is quadratic $O(n^2)$
  - in the worst case, for each *i* we do *(i-1)* swaps inside the inner for-loop
  - therefore, overall number of swaps (when *i* goes from 2 to n in the outer for-loop) is

$$T(n)=1+2+3+\ldots+(i-1)+\ldots.+n-1 = n*(n-1)/2$$

# Analysis: Insertion Sort Algorithm

- Unlike *selection-sort*, complexity of *insertion-sort* DOES depend on specific instance of the problem (data values)

**Exercise**: show that the *best case* complexity is O(n)

(consider the case when the array is already sorted)

- also, works well also if array is “almost” sorted
- however, **average case complexity will be still O($n^2$)**

# Radix Sort

- Sorts objects based on some *key* value found within the object
- Most often used when keys are strings of the same length, or positive integers with the same number of digits
- Uses queues; does not sort "in place"
- Other names: *postal sort*, *bin sort*

# Radix Sort Algorithm

- Suppose keys are *k-digit* integers
- Radix sort uses an array of 10 queues, one for each digit 0 through 9
- Each object is placed into the queue whose index is the least significant digit (the 1's digit) of the object's key
- Objects are then dequeued from these 10 queues, in order 0 through 9, and put back in the original queue/list/array container; they're sorted by the last digit of the key

# Radix Sort Algorithm

- Process is repeated, this time using the 10’s digit instead of the 1’s digit; values are now sorted by last two digits of the key
- Keep repeating, using the 100’s digit, then the 1000’s digit, then the 10000’s digit, …
- Stop after using the most significant ($10^{n-1}$’s ) digit
- Objects are now in order in original container

# Algorithm: Radix Sort

```
Assume n items to be sorted, k digits per key, and t possible values
for a digit of a key, 0 through t-1. (k and t are constants.)
For each of the k digits in a key:
  While the queue q is not empty:
    Dequeue an element e from q.
    Isolate the kth digit from the right in the key for e; call it d.
    Enqueue e in the dth queue in the array of queues arr.
  For each of the t queues in arr:
    While arr[t-1] is not empty
      Dequeue an element from arr[t-1] and enqueue it in q.
```

# Radix Sort Example

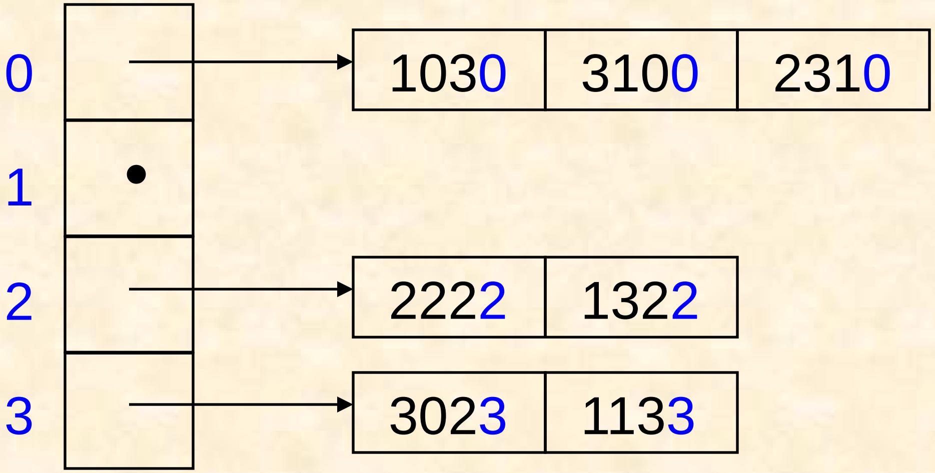
Suppose keys are 4-digit numbers using only the digits 0, 1, 2 and 3, and that we wish to sort the following queue of objects whose keys are shown:

| | | | | | | |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 2222 | 1322 | 3100 | 1133 | 2310 |

# Radix Sort Example

| | | | | | | |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 2222 | 1322 | 3100 | 1133 | 2310 |

*First* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item's last digit



0
1030 3100 2310

1

2
2222 1322

3
3023 1133

Array of queues after the *first* pass

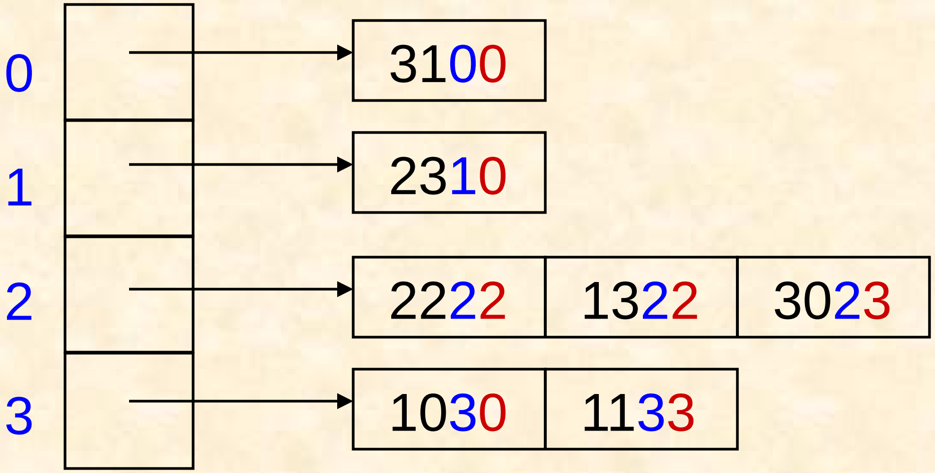
Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one):

| | | | | | | |
|---|---|---|---|---|---|---|
| 1030 | 3100 | 2310 | 2222 | 1322 | 3023 | 1133 |

# Radix Sort Example

| | | | | | | |
|---|---|---|---|---|---|---|
| 1030 | 3100 | 2310 | 2222 | 1322 | 3023 | 1133 |

*Second* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item’s 2nd last digit


0
3100
1
2310
2
2222
1322
3023
3
1030
1133

Array of queues after the *second* pass

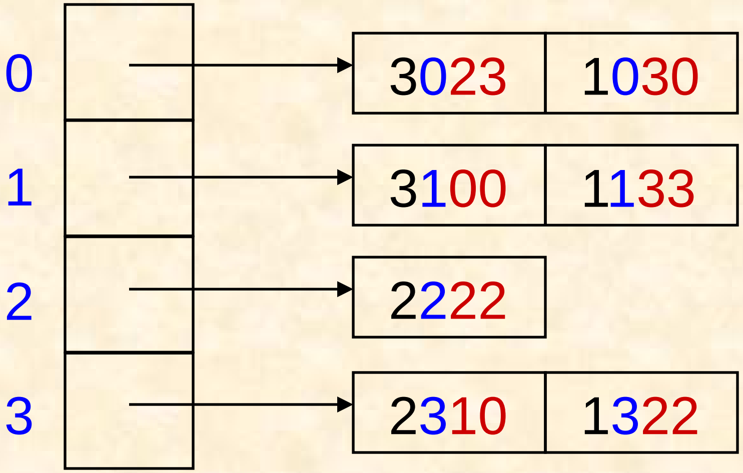
Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one):

| | | | | | | |
|---|---|---|---|---|---|---|
| 3100 | 2310 | 2222 | 1322 | 3023 | 1030 | 1133 |

# Radix Sort Example

| | | | | | | |
|---|---|---|---|---|---|---|
| 3100 | 2310 | 2222 | 1322 | 3023 | 1030 | 1133 |

*First* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item's 3rd last digit



0 → 3023, 1030

1 → 3100, 1133

2 → 2222

3 → 2310, 1322

Array of queues after the *third* pass

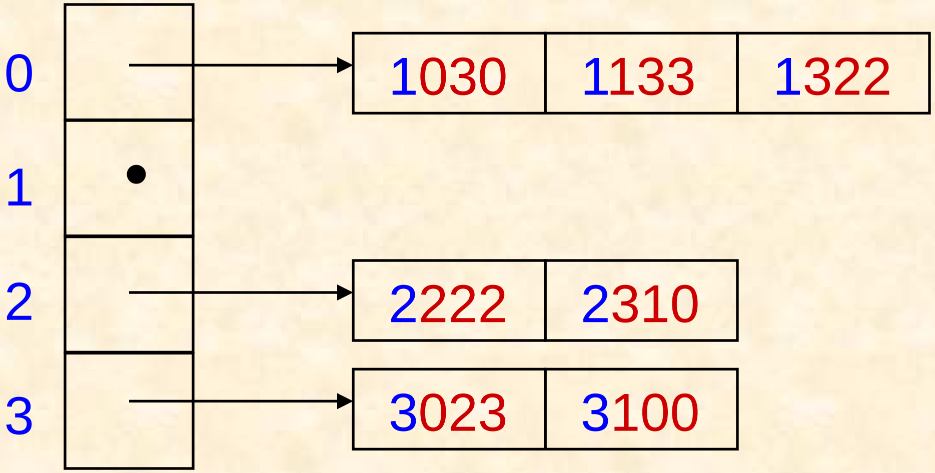
Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one):

| | | | | | | |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 3100 | 1133 | 2222 | 2310 | 1322 |

# Radix Sort Example

| | | | | | | |
|---|---|---|---|---|---|---|
| 3023 | 1030 | 3100 | 1133 | 2222 | 2310 | 1322 |

*First* pass: while the queue above is not empty, dequeue an item and add it into one of the queues below based on the item's first digit



Array of queues after the *fourth* pass

Then, items are moved back to the original queue (first all items from the top queue, then from the 2nd, 3rd, and the bottom one): NOW IN ORDER

| | | | | | | |
|---|---|---|---|---|---|---|
| 1030 | 1133 | 1322 | 2222 | 2310 | 3023 | 3100 |

# Analysis: Radix Sort

- We’ll count the total number of enqueue and dequeue operations
- Each time through the outer *for* loop:
    - In the *while* loop: n elements are dequeued from q and enqueued somewhere in arr: 2*n operations
    - In the inner *for* loop: a total of n elements are dequeued from queues in arr and enqueued in q: 2*n operations

# Analysis: Radix Sort

- So, we perform 4*n enqueue and dequeue operations each time through the outer loop
- Outer for loop is executed k times, so we have 4*k*n enqueue and dequeue operations altogether
- But k is a constant, so the time complexity for radix sort is O(n)

  - COMMENT: If the maximum number of digits in each number k is considered as a parameter describing problem input, then complexity can be written in general as O(n*k) or O(n*log(max_val))