

Topic 1: Numeric, class, and array data types

1. Collections and arrays
2. Design: divide and conquer
3. Algorithm verification
4. Searching and sorting
5. Algorithm complexity

1. Collections and arrays

- General-purpose collection \approx dictionary \approx dynamic set
- Cellular implementations (random access):
 - arrays
 - random-access files
 - vectors
- Linked-list implementation:
 - access is sequential (must traverse all previous items, as in text file)
- Later: branching structures (trees)

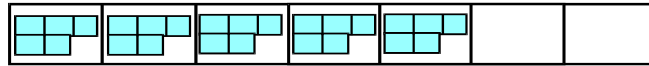
General-purpose collection

- A kind of dynamic set or collection that implements
 - insertion
 - deletion
 - search
- The attribute or field of an element of a collection through which we access the element is called the *key*

Dynamic set:

- A set that may grow, shrink, or be updated
- *Note:* a set is defined as an aggregate of distinct items, real or imaginary; operations include union, intersection, and difference
- Example of a dynamic set: a database relation

Arrays use random access



- Program may instantly store or retrieve data
- Components of array or file are found by calculating their offset from the first one
- We may implement a collection as an array or random file of structures or objects

A collection: many items of one type

```

class score_colx
{
    enum {MAX_SCORES = 6};
public:
    score_colx()
        { size = 0; };
    bool append(int new_elem);
    void input();
    void display();
private:
    int element[MAX_SCORES]; // Keyboard-input member function.
    int size;
};

void main()
{
    score_colx list;
    list.input();
    list.display();
}

void score_colx::input()
{
    int input_value;
    do {
        cout << "Score (-1 to quit):";
        cin >> input_value;
    } while (input_value >= 0 &&
            append(input_value));
}
    
```

An instance of *score_lists*

size 5

element 2 15 9 4 32

A collection of int

Managing a collection

```
bool score_colx::append(int new_element)
// Adds <new_element> to collection.
{
    bool success = (size < MAX_SCORES);
    if (success)
        element[size++] = new_element;
    else
        cout << "score_colx::append: "
              << "collection already full" << endl;
    return success;
}
```

```
void score_colx::display()
// Displays values.
{
    cout << endl;
    for (int i = 0; i < size; ++i)
        cout << element[i] << endl;
}
```

*Array
boundary
check to see if
there's room*

Computation with collections is not necessarily algorithmic

- A collection object may be searched, added to, or deleted from without end
- The client that sends messages to the object determines the sequence of messages
- The sequence of inputs and outputs in the life of an object is a stream
- This stream expresses the interaction of the object with its environment

Collection class in C++

```

class employees
{
public:
    void display() { cout << ID << " " << name; };
private: int ID;    char name[40];
};
    
```

```

class emp_collections
{
public:
    void display()
        { for (int i=0; i < size; i++)
          emp[i].display(); }
private:
    employees emp[100];
    int size;
};
    
```

David Keil
Data Structures 1. Arrays
6/08
9

A collection of employees

```

class rosters
{
public:
    rosters() { num_recs = 0; }
private:
    employees emp[100];
    int num_recs;
};
    
```

```

struct employees
{
    char name[40];
    int salary;
};
    
```

A collection class

[payroll.cpp]

An instance of collection class rosters

size

5

item

David Keil
Data Structures 1. Arrays
6/08
10

2. Design: Divide and conquer

- *Tip-off*: Can use divide-and-conquer if an instance of the problem either has a simple solution, or can be broken down into two simpler problems
- *Example*: Reversing array is simple for input of size 1; for larger input, just break the array into *first* and *remainder*, one of which is simple to reverse and the other is simpler than the original problem
- *Recursion* uses divide-and-conquer

Data structures and algorithm design

- Problem specification often includes storage of data in a collection
- For any collection structure, certain *operations* apply, implemented by *algorithms*
- Choice of structure is determined by performance of these algorithms, including scalability as size of input rises

Base case and recursive case

- *Recursive method*: one that calls itself
- *Base case*: If input or argument to a recursive function is in a certain range, a simple solution is immediately returned
- *Recursive case*: If base case fails to apply, the entire algorithm or function is executed again on a simpler value than the argument

The recursive factorial method

```
int factorial(int n)
{
    if (n <= 1) Base case
        return 1;
    else Recursive call
        return n * factorial(n - 1)
}
```

The factorial of a natural number is the product of all the natural numbers from 1 to the number

Adding without a “+” table

```
int sum(int a, int b)
// Returns sum of a and b.
{ // Precondition: a >= 0
  if (a == 0)
    return b;
  else
    return sum(a-1, b+1);
}
```

- What is base case? Recursive case?
- How are infinite recursion and stack overflow avoided?

Exponentiation using recursion

- *Problem:* write a function with *unsigned int* parameters, *base* and *exponent*, that returns the value $base^{exponent}$
- *Pseudocode for solution:*
if exponent is 0
 $power(base, exponent) = 1$
otherwise
 $power(base, exponent) = base \times power(base, exponent - 1)$
- Divide-and-conquer has broken the problem into two simpler ones; the second invokes the *power* algorithm itself

Pros and cons of recursion



- Solution is concise
- Solution may be easy to analyze for time performance (see topic 3)



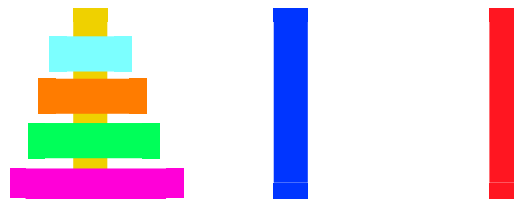
- Recursion uses call stack; deep recursion may overflow stack
- Recursion entails overhead of a function call



- In balance: recursion is useful in design, often less useful in coding

Towers of Hanoi/Brahma

- Disks start all stacked by size on one of 3 pegs
- Objective: move all disks from first peg to third peg
- Rule: move one at a time, never place a disk on a smaller disk



Solution to Hanoi (informal)

Divide and conquer for n disks:

1. Move the top $(n - 1)$ disks to the intermediate peg.
2. Move bottom disk to the final peg
3. Move all disks on the intermediate peg to final peg.

Towers of Hanoi pseudocode

Hanoi(source, intermed, destination, disks):

If $disks = 1$

Move disk on *source* to *destination*

otherwise

Hanoi(source, destination, intermed, disks - 1)

Move top of *source* to *destination*

Hanoi(intermed, source, destination, disks - 1)

- This algorithm is recursive
- *disks* is the number of disks to move
- *source*, *intermed*, and *destination* are pegs (1, 2, or 3)

A rabbit problem

(Leonardo Pisano Bigollo,
1202)



“Suppose you have a pair of rabbits and suppose every month each pair bears a new pair that from the second month on becomes productive. How many pairs will you have in a year?”

(G. Rawlins, *Compared to What?*, p. 31)

Recursive solution to rabbit problem: *Fibonacci*

Month	0	1	2	3	4	5
Pairs	1	1	2	3	5	8

- $Pairs(month) = \begin{cases} 1 & \text{if } month \leq 1 \\ Pairs(month - 1) + Pairs(month - 2) & \text{otherwise} \end{cases}$
- **Challenge:** How many *additions* does it take to compute $Fib(n)$?

Linear vs. nonlinear recursion

- Double recursion as in *Fibonacci* is nonlinear
- Linear (single) recursion and iteration are faster
- Note: base case in recursive *fib* returns 1, so no matter how big *n* is, we build *fib(n)* 1 at a time

```

int lrfib(int n,int& prev)          int ifib(int n)
// Linear-recursive Fibonacci.     // Iterative Fibonacci using
{                                  // dynamic programming.
  count++;                          {
  if (n < 2)                          count++;
  {                                  int fib[1000];
    prev = (n == 1 ? 1 : 0);          fib[0] = fib[1] = 1;
    return 1;                          for (int i = 2; i <= n; ++i)
  }                                  {
  else                                  count++;
  {                                  fib[i] = fib[i-1] + fib[i-2];
    int prevprev;                      }
    prev = lrfib(n-1,prevprev);        return fib[n];
    return prev + prevprev;          }
  }
}

```

3. Algorithm verification

- Testing can detect errors but cannot assure correctness
- Desk checking helps verify
- Assertions help verify: preconditions, postconditions, loop invariants

White-box vs. black-box testing

- In white-box (unit) testing we may look at the code and design test data that exercises it
- *Problem:* We may tailor our test data to prove that the code works
- In black-box (QA) testing, the test designers do not see the code



- *Problem:* How can testing exercise possibly vulnerable parts of code if test-design team can't see the code?
- Exhaustive black-box testing require test of all possible input values; desk checking is recommended

Assertion:

- A true/false proposition that should be true at a certain point in an algorithm's execution, if the algorithm is correct.
- The *assert* macro in *assert.h* supports debugging assertions

```
float quotient(int a,int b)
// Returns (a/b).
// Terminates prog. If b=0.
{
    assert(b != 0, "Division by zero");
    return (float)a / b;
}
```

Assertions and correctness

- A comment that is an assertion tells not what *occurs*, but something about *values* of variables and expressions
- Valid assertions can help us establish that our code does what we claim

Conditions for software correctness

- Program eventually halts under all conditions
- Thus, values that control termination of a loop must *converge*
- For *all* possible inputs or parameter values, a program or function must have a correct result
- Mental and documentation tools to establish correctness: preconditions, postconditions, loop invariants



Comments can show correctness

```

void main()
{
    int score[] = { 2,5,18,4,9,12,10 };
    clear(score,4);
    for (int i=0; i < 7; ++i)
        cout << score[i] << " ";
}
void clear(int A[],int high)
// Zeros the elements of <A> from 0 to <high>
{
    // Precondition: 0 <= high < (# elements in A)
    int i = high;
    while (i >= 0)
        // Loop invariant: either i = high, or
        // A[i+1 ... high] are all 0's.
        A[i--] = 0;
    // Postcondition: A[0...high] are all 0's.
}

```

[clear.cpp]

Output: 0 0 0 0 0 12 10

- **Precondition:** An assertion that should be true before a certain sequence of steps in an algorithm
- **Postcondition:** An assertion that should be true at the end of a certain sequence
- *Example:*
Algorithm to add a series of numbers.
 - Precondition: *total* is 0
 - Postcondition: *total* stores the sum of all input values

Loop invariant:

An assertion, about the state of a program or function, that is true at the start of each iteration of a loop, helping to establish the validity of a postcondition

Rationale: If we can show that an assertion is true at the top of the loop and true throughout its execution, then we can show that the assertion is a postcondition that is true after the loop terminates

Insertion in a sorted array

Problem:

- Write code to insert value x into a sorted array of integers, A , of size m , maintaining it in ascending order

2 →

1	3	4	6	
---	---	---	---	--

1	2	3	4	6
---	---	---	---	---

- Use preconditions, postconditions, and loop invariants to prove correctness of your algorithm

Array-insertion code

```
void insert(double A[], int& n_elts, double x)
{
// Preconditions: A not full; A is ascending
int i = n_elts;
while (x < A[i - 1] && i > 0)
{ // LI: x is less than any value
  // in A[i-1 .. n_elts-1]
  A[i] = A[i-1]; // Move elements greater than
                // new_item to the right
  ++i;
}
A[i] = x; // Drop new_item in place
++n_elts;
// Postconditions: A contains 'x'
// A is ascending, size of A is 'n_elts'.
}
```

4. Searching and sorting arrays

- Essential for efficient access to collections
- Here we assume array implementation of collections
- Search algorithms: linear, binary
- Sorting algorithms: selection, Bubble, insertion, Quick

Linear search

for element with value x in array A

Search(A, x)

$i \leftarrow 1$

$found \leftarrow false$

while $found = false$ and $i \leq size[A]$

 if $A[i]$ matches x

$found \leftarrow true$

$i \leftarrow i + 1$

return value of $found$

Searching a collection of structures

- The search key is one member of the structure type
- To find the name of employee whose ID is x , for example:

```
for (int i=0; i < n; ++i)
    if (A[i].ID == x)
        return A[i].name;
```

Finding string length by search

```

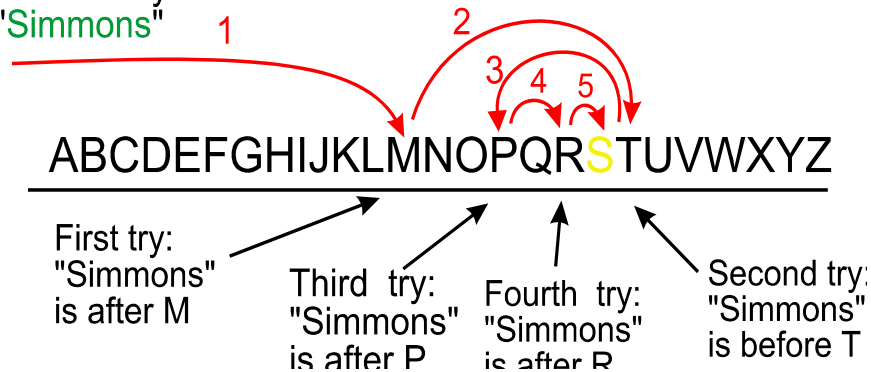
int i=0,len=0,found = 0;
char name[80];

printf("Enter name: ");
scanf("%s",&name);
while(! found)
{
    if (name[i] == '\0')
    {
        len = i;
        found = 1;
    }
    ++i;
}
printf("Length = %i\n",len); [leng_str.c]

```

Binary phone-book search

Search key:
"Simmons"



- Each step eliminates half the unsearched data, cuts remaining work in half

Binary search algorithm

Finds value *key* in subarray [*first...last*] of array *A*, or reports failure

```

if first > last (i.e., nothing to search)
  return false
otherwise
  middle ← (first + last) ÷ 2
  if A[middle] matches key
    return true
  otherwise
    if A[middle] > key
      return Bin-search(A, first, middle - 1, key)
    otherwise
      return Bin-search(A, middle + 1, last, key)

```

Insertion-sort(*A*)

Precondition: *A* is an array

num_sorted ← 1

Repeat

Invariant: *A*[1...*num_sorted*] is ascending

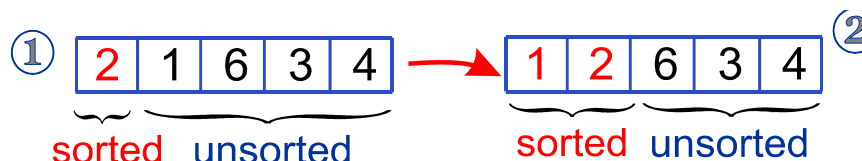
i ← *num_sorted* + 1

Array-insert (*A*, *num_sorted*, *A*[*i*])

num_sorted ← *num_sorted* + 1

until *num_sorted* is size of *A*

Postcondition: *A* is sorted ascending



Bubble sort (*informal*)

Repeat
 for each element up to $A[n - 1]$
 if it is greater than its successor
 swap them
 until none were found out of order

- The exit condition proves that bubble sort works if it halts
- How do we know it halts?

Bubble-sort(A)

Precondition: A is an array

$num_passes \leftarrow 0$

Repeat

Invariant: Rightmost (num_passes) elements are sorted; if *swapped* is false and $num_passes > 0$, then A is sorted

$swapped \leftarrow false$

for $i \leftarrow 1$ to $size(A) - 1$

Invariant: $A[i]$ is not smaller than any in $A[0..i-1]$

if $A[i] > A[i + 1]$

$swap(A[i], A[i + 1])$

$swapped \leftarrow true$

$num_passes \leftarrow num_passes + 1$

until *swapped* is false

Postcondition: A is sorted ascending

Selection sort

```

void main()
{
    int score[] = {3,2,9,7,4,6,5,1,8,0};
    int n = sizeof(score)/sizeof(int);
    selection_sort(score,n);
    for (int i = 0; i < n; ++i)
        cout << score[i] << " ";
}

void selection_sort(int A[],int size)
{
    for (int i=0; i < size-1; ++i)
    {
        // Find min { A[i..size-1]}, swap w/ A[i]:
        int smallest = i; // Index of least value
        for (int j = i+1; j < size; ++j)
            if (A[j] < A[smallest])
                smallest = j;
        swap(A[i],A[smallest]);
    }
}

```



Steps:

21634

12634

12634

12364

12346

12346

Quicksort

- C. A. R. Hoare, UK, ca. 1950
- Recursive
- Faster by far than the above algorithms
- Harder by far to understand
- Uses algorithm *Partition* $A[m..n]$ into left and right subarrays using some value in $A[m..n]$ as a pivot. Values less than pivot go to its left, others to its right

Quicksort code

```
void QuickSort(float A[],int lo,int hi)
// Sorts subarray A[lo..hi] ascending.
{
    if (lo < hi)
    {
        int pivotloc = partition(A,lo,hi);
        QuickSort(A, pivotloc+1, hi);
        QuickSort(A, lo, pivotloc-1);
    }
}
```

Subscripts

*Recursively sort left
and right partitions*

*Find arbitrary pivot value,
group smaller A elements
to its left, greater to right*

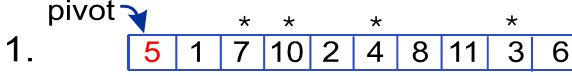
Partition code

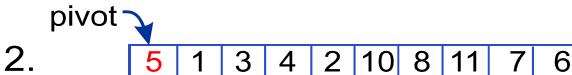
Partitions subarray *num[first..last]*, returns index of pivot value

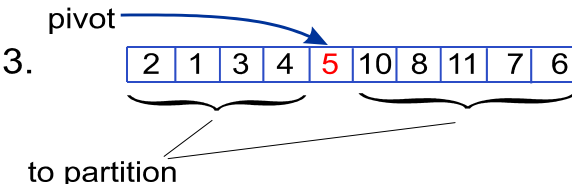
```
int partition(int num[], int first, int last)
{
    int pivot = num[first];
    int left = first + 1, right = last;
    do {
        while (left <= right && num[left] <= pivot)
            ++left;
        while (left <= right && num[right] > pivot)
            --right;
        if (left < right)
            swap (num[left], num[right]);
    } while (left <= right);
    swap (num[first], num[right]);
    return right;
}
```

Move pivot into middle

The partition step

1.  *Repeatedly finds leftmost element in left partition that should be in right and*

2.  *rightmost that should be in left, swaps them*

3.  *Here, 3 swaps with 7, 10 swaps with 4, and pivot 5 swaps into place*

David Keil Data Structures 1. Arrays 6/08 47

Choosing the pivot

- Remember, pivot's *location* is a subscript, pivot *itself* is an array-element value
- You may choose any element as a pivot, including one from a random location
- Two commonly used pivot locations are the first element in a partition, and the middle one
- Best pivot is one that is about the median value in the array, leaving equal-sized partitions

5. Algorithm complexity

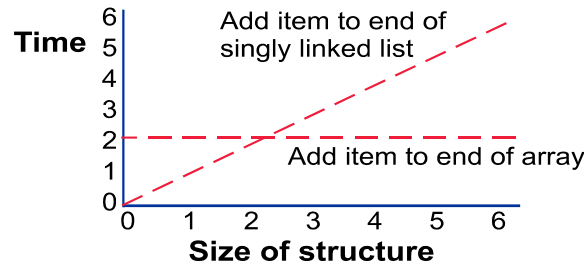


- For a given algorithm, we can find a formula for space or time consumed, in terms of the quantity of data processed
- We abstract hardware details such as instructions per pixel drawn or clock speed
- Time is often more scarce than space

Resource-related problems

1. How much storage is required for sales figures for 365 days, 2100 stores, 583 items?
 d days, s stores, i items?
2. How many executions of machine-language instructions are needed to display a solid square, 300 pixels on a side, at k instructions per pixel?
A square, p pixels on a side?
3. How many guesses are needed to guess a 4-digit PIN number? An n -letter password?

Time complexities of algorithms may be expressed as *functions*



- More data may require more time to process
- These functions are *not* the same as those calculated by the algorithms we analyze
- Our goal is to analyze running time independent of hardware and data size

Why we study complexity

- Designers and programmers must choose algorithms and data structures to solve problems
- Different data structures and algorithms are adapted to different problems
- The “best” data structure is often the one with the best balance of time complexities among the operations on the data structure
- E.g., insert, delete, search, traverse

Time complexity

- A linear problem ($T(n) = k n$):
Find the largest element of an unsorted array
- A quadratic problem ($T(n) = k n^2$): Show results of comparing n items, a pair at a time
- An exponential problem ($T(n) = k 2^n$):
Show all possible n -bit values

>	'b'	'c'	'a'
'b'	F	F	T
'c'	T	F	T
'a'	F	F	F

Tools of algorithm analysis

1. Empirical performance testing

- timer
- counter

2. Theoretical tools

The tools of *algorithm analysis*
(see 63.347)

- Big-O notation
- recurrences

Your project's result data

Number of search steps (or milliseconds)

# of <u>records(n)</u>	<u>Unsorted</u> <u>array</u>	<u>Sorted</u> <u>array</u>	<u>BST</u>	<u>Hash</u> <u>table</u>
10				
100				
300				
1000				
5000				
10000				

(sample test-results grid)

Counting steps in a program

```

unsigned count = 0;                                     [timed_bin_search.cpp]
void main() { ... }
bool binary_search(int A[],int key,int size)
{
    int first = 0, last = size-1, middle;
    bool found = false;
    while (first <= last && ! found)
    {
        middle = (first + last) / 2;
        count++; // We define a step as a comparison
        if (A[middle] == key) found = true;
        else if (key < A[middle]) last = middle - 1;
        else first = middle + 1;
    }
    return found;
}

```

```

Enter search key: 4 Found
12 elements 2 comparisons
Enter search key: 5 Not found
12 elements 3 comparisons

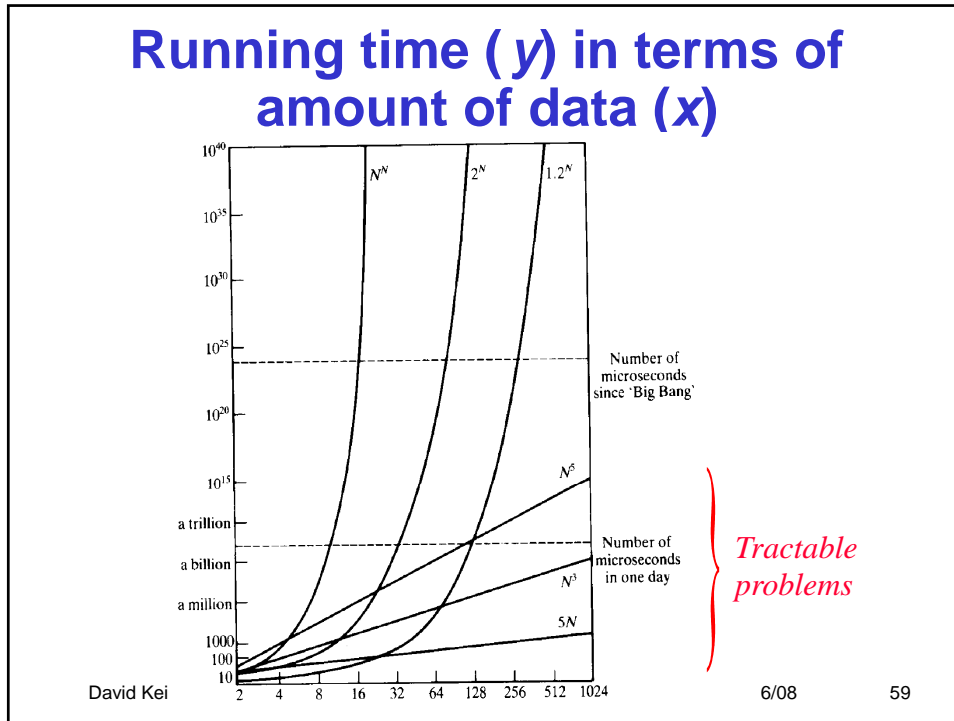
```

Introducing big-O notation

- Expresses the time efficiency of an algorithm as a *function* of amount of data processed
- *Example:* To say that an algorithm is $O(n)$ means roughly that the algorithm's running time is proportional to the size of the data set

Grouping time functions

- Functions that grow roughly at the same rate are in the same Big-O category
- We group together all constant functions, linear functions, quadratic functions, etc., as $O(1)$, $O(n)$, $O(n^2)$
- Logarithmic time: $O(\log_2 n) = O(\lg n)$
- Quadratic time: $O(n^2)$
- Exponential time: $O(2^n)$



Complexity of array insertion

Insert-ascending (A, last, new-value)

```

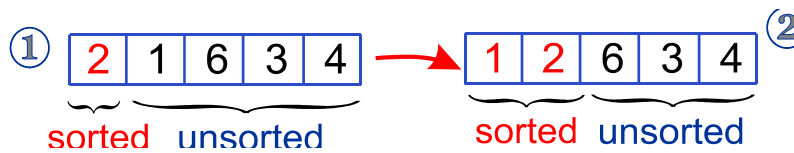
i ← last
while i > 1 and new-value < A[ i ]
    A[ i ] ← A[ i - 1 ]
    i ← i - 1
A[ i ] ← new-value
last ← last + 1
    
```

Inserts new-value at its proper location in ascending sequence of A's elements in 1...last

- The number of iterations of the loop determines algorithm's time complexity
- Amount of data (n) is last; loop iterates up to n times
- $T_{\text{Insert}}(n) = O(n)$

Insertion-sort(A) $num_sorted \leftarrow 1$ While $num_sorted < size(A)$

$$Insert\text{-}ascending(A, num_sorted,$$

$$A[num_sorted + 1])$$
 $num_sorted \leftarrow num_sorted + 1$ *Complexity analysis: $O(?)$ Why?***Bubble sort**

Repeat

 $swap \leftarrow false$ for $i \leftarrow 1$ to $size(A) - 1$ if $A[i] > A[i + 1]$ $swap(A[i], A[i + 1])$ $swap \leftarrow true$ until $swap = false$

- The use of nested loops suggests what about running time, for an n -element array?
- *Challenge:* Write a recursive version of *Bubble*

Selection Sort, recursive version

Selection-sort (A , $start$, $size$)

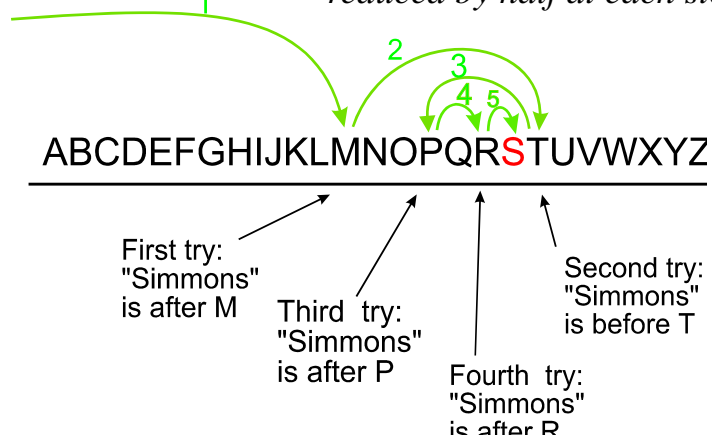
If $size > 1$

1. Find lowest value in $A[start..size]$
2. Swap it with $A[start]$
3. **Selection-sort** (A , $start + 1$, $size$)

Binary (phone-book) search

Search key:
"Simmons"

*The remaining work is
reduced by half at each step*



Binary search (A, first, last, key)

```
if  $first \leq last$ 
   $middle \leftarrow (first + last) \div 2$ 
  if  $key = A[middle]$ 
    return true
  if  $key < A[middle]$ 
    return Binary-search(A, first,
                            $middle - 1$ , key)
  otherwise
    return Binary-search(A, last,
                            $middle + 1$ , key)
otherwise
  return false
```

Loops and complexity

- A single loop of n iterations, $O(1)$ each, is $O(n)$
- A loop nested to two levels, each with roughly n iterations, $O(1)$ each, is $O(n^2)$
- If we start with n items to look at and cut our remaining work in half at each step, then the job will take $O(\log_2 n)$ such steps.
- If our loops are nested to n levels, as in password guessing, then algorithm is $O(2^n)$ — offer job to someone else

Average-case analysis

- We are looking for *expected* time
- This is the average running time over all equally probable data sets
- *Example:* All permutations of 50 integers in range 0..9 are equally probable as contents of a random 50-element array of 1-digit numbers
- Statistical tools include random variables, combinatorics (see discrete math)

Average case and worst case

- Big-O result for average case *tends* to match worst-case analysis
- *Example:* in linear search, possible running times range from 1 to n steps
- *Linear search:*
 - worst case is $O(n)$;
 - best case is $O(1)$;
 - average case is $(n + 1) / 2$ steps ($O(n)$)
- *Exception:* Quicksort worst case is $O(n^2)$, average case is $O(n \lg n)$

Complexity trade-offs with data structures

- Classical operations on a collection:
insert, delete, search, traverse
- Some data structures support faster algorithms for certain of these operations
- We usually select a data structure for an application depending on which operations are time-critical
- Using more space may permit faster access to data

Terminology (correctness)

algorithm	nonlinear recursion
<i>assert</i> macro	object
assertion	object-oriented design
base case	partial correctness
class	postcondition
collection	precondition
data abstraction	recursion
divide-and-conquer algorithm	recursive case
induction	termination
linear recursion	Towers of Hanoi problem
loop invariant	

Terminology (complexity)

algorithm complexity	logarithmic time
asymptotic analysis	nested loop
average case	$O(\log n)$
best case	$O(n)$
big-O notation	$O(n^2)$
binary search	omega (Ω)
bubble sort	partition
dominant term	quadratic time
exponential time	Quicksort
insertion sort	running time
linear and nonlinear	selection sort
recursion	space complexity
linear search	worst case

References

Drozdek and Simon. *Data Structures*.

Bertrand Meyer. Writing correct software.

DDJ, ____