Searching & Sorting

Dr. Chris Bourke
Department of Computer Science & Engineering
University of Nebraska—Lincoln
Lincoln, NE 68588, USA
http://cse.unl.edu/~cbourke
cbourke@cse.unl.edu

2015/04/29 12:14:51

These are lecture notes used in CSCE 155 and CSCE 156 (Computer Science I & II) at the University of Nebraska—Lincoln.

Contents

List of Algorithms

Code Samples

List of Figures

1. Overview

1.1. CSCE 155E Outline

1. Introduction
2. Linear Search (basic idea, example, code, brief analysis)
3. Binary Search (basic idea, example, code, brief analysis, comparative analysis)
4. Bubble Sort (Basic idea, example, code, brief analysis)
5. Selection Sort (Basic idea, example, code, brief analysis)
6. Quick Sort (Basic idea, example, comparative analysis only)
7. Function pointers
8. Sorting & Searching in C
9. Honors: Comparators, Searching, Sorting in Java

1.2. CSCE 156 Outline

1. Introduction
2. Linear Search (basic idea, pseudocode, full analysis)
3. Binary Search (basic idea, pseudocode, full analysis, master theorem application, comparative analysis)
4. Bubble Sort (Basic idea, example, pseudocode, full analysis)
5. Selection Sort (Basic idea, example, pseudocode, full analysis)
6. Insertion Sort (Basic idea, example, pseudocode, full analysis)
7. Quick Sort (Basic idea, example, pseudocode, full analysis)
8. Merge Sort (Basic idea, example, pseudocode, full analysis)
9. Sorting Stability
10. Comparators in Java
11. Equals & Hash Code in Java
12. Searching in Java
13. Sorting in Java

Part I.
Searching

2. Introduction

Problem 1 (Searching)**.**
Given: *a collection of elements,* $A = \{a_1, a_2, \ldots, a_n\}$ *and a* key *element* e_k
Output: *The element* a_i *in* A *that matches* e_k

Variations:

- Find the first such element
- Find the last such element
- Find the index of the element
- Key versus "equality" based
- Find all such elements
- Find extremal element(s)
- How to handle failed searches (element does not exist)?

Note: the problem is stated in general terms; in practice, searching may be done on arrays, lists, sets, or even *solution spaces* (for optimization problems).

3. Linear Search

- A basic and straightforward solution to the problem is the *linear search algorithm* (also known as sequential search).
- Basic idea: iterate over each element in the collection, compare with the key e_k

3.1. Pseudocode

```
INPUT  : A collection of elements A = {a_1, ..., a_n} and a key e_k
OUTPUT: An element a ∈ A such that a = e_k according to some criteria; φ if no
        such element exists
1 FOREACH a_i ∈ A DO
2 |   IF a_i = e_k THEN
3 |   |   output a_i
4 |   END
5 END
6 output φ
```

Algorithm 1: Linear Search

3.2. Example

Consider the array of integers in Table 1. Walk through the linear search algorithm on this array searching for the following keys: 20, 42, 102, 4.

index	0	1	2	3	4	5	6	8	9
element	42	4	9	4	102	34	12	2	0

Table 1: An array of integers

3.3. Analysis

As the name suggests, the complexity of linear search is linear.

1. Input: the collection of elements A
2. Input size: n as there are n elements
3. Elementary Operation: the comparison to the key in line 2
4. Best case: we immediately find the element, $O(1)$ comparisons
5. Worst case: we don't find it, or we find it at the last elements, n comparisons, $O(n)$
6. Average case (details omitted): an expected number of comparisons of $\frac{n}{2} \in O(n)$

4. Binary Search

A much more efficient way to search is the *binary search* algorithm. The basic idea is as follows. Under the assumption that the collection is *sorted*, we can:

- Examine the middle element:

1. If the the middle element is what we are searching for, done
2. If the element we are searching for is less than the middle element, it must lie in the lower-half of the list
3. Otherwise, it must lie in the upper-half of the list

- In either case: list is cut in half (at least); we can repeat the operation on the sub-list
- We continue until either: we find the element that matches the key, or the sub-list is empty

4.1. Pseudocode: Recursive

INPUT : A *sorted* collection of elements $A = \{a_1, \ldots, a_n\}$, bounds $1 \leq l, h \leq n$, and a key e_k

OUTPUT: An element $a \in A$ such that $a = e_k$ according to some criteria; ϕ if no such element exists

```
1  IF l > h THEN
2  |  output φ
3  END
4  m ← ⌊(h+l)/2⌋
5  IF a_m = e_k THEN
6  |  output a_m
7  ELSE IF a_m < e_k THEN
8  |  BinarySearch(A, m + 1, h, e)
9  ELSE
10 |  BinarySearch(A, l, m − 1, e)
11 END
```

Algorithm 2: Binary Search – Recursive

4.2. Pseudocode: Iterative

INPUT : A *sorted* collection of elements $A = \{a_1, \ldots, a_n\}$ and a key e_k
OUTPUT: An element $a \in A$ such that $a = e_k$ according to some criteria; ϕ if no such element exists

```
1  l ← 1
2  h ← n
3  WHILE l ≤ h DO
4  |  m ← ⌊(h+l)/2⌋
5  |  IF a_m = e_k THEN
6  |  |  output a_m
7  |  ELSE IF a_m < e_k THEN
8  |  |  l ← (m + 1)
9  |  ELSE
10 |  |  r ← (m − 1)
11 |  END
12 END
13 output φ
```

Algorithm 3: Binary Search – Iterative

4.3. Example

Consider the array of integers in Table 2. Walk through the linear search algorithm on this array searching for the following keys: 20, 0, 2000, 4.

index	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
element	0	2	4	4	9	12	20	21	23	32	34	34	42	102	200	1024

Table 2: A sorted array of integers

4.4. Analysis

1. Input: collection of elements, A
2. Input size: n, the number of elements
3. Elementary Operation: comparison, performed once for each "middle" element
4. Analysis:
   - Let $C(n)$ be the (maximum) number of comparisons made by binary search on a collection of size n

- On each recursive call, one comparison is made, plus the number of comparisons on the next recursive call
- The list is cut in half; so the next recursive call made will contribute $C\left(\frac{n}{2}\right)$
- In total:
$$C(n) = C\left(\frac{n}{2}\right) + 1$$
- By case 2 of the Master Theorem, Binary search is $O(\log n)$

5. Another view:
   - We begin with a list of size n
   - After the first iteration, it is halved:
   $$\frac{n}{2}$$
   - After the second, it is halved again:
   $$\frac{n}{2^2}$$
   - After i such iterations:
   $$n \cdot \underbrace{\frac{1}{2} \cdot \frac{1}{2} \cdot \cdots \cdot \frac{1}{2}}_{i} = \frac{n}{2^i}$$
   - The algorithm terminates when the list is of size 1 (actually, zero, but we'll treat the last iteration separately):
   $$\frac{n}{2^i} = 1$$
   - Solving for $i = \log n$, thus $O(\log n)$ iterations

Contrast binary search with linear search: suppose we wanted to search a database with 1 trillion (10^{12}) records.

- Linear search: approximately 10^{12} comparisons required
- Binary search: approximately $\log_2(10^{12}) \approx 40$ comparisons

Suppose further that a list initially has n elements and its size is doubled; then

- Linear search: twice as many more comparisons on the new list
- Binary search: $\log_2(2n) = \log_2(2) + \log(n) = \log(n) + 1$; that is, only one more comparison than before
- Binary search requires an up-front, $O(n \log n)$ cost to sort (or less if an order is *maintained*)
- If only done once, no need to sort, just use linear search
- If repeated, even a small amount, $O(\log n)$ searches say, then it pays to sort and use binary search

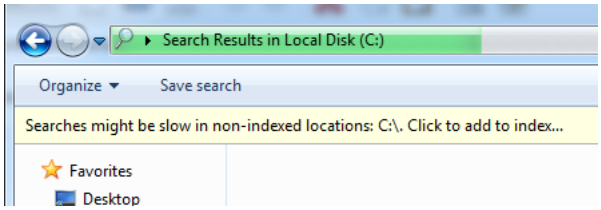

Figure 1: Illustrative example of the benefit of ordered (indexed) elements, Windows 7

5. Searching in Java

5.1. Linear Search

Code Sample 1: Java Linear Search for integers

```
1  /**
2   * This function returns the index at which the given element
3   * first appears in the given array.  Returns -1 if the array
       does
4   * not contain the element
5   */
6  public static int linearSearch(int a[], int element)
7  {
8    for(int i=0; i<a.length; i++){
9      if(a[i] == element){
10       return i;
11     }
12   }
13   return -1;
14 }
```

5.2. Binary Search

Code Sample 2: Java Recursive Binary Search for integers

```
1  /**
2   * This function returns an index at which the given element
3   * appears in the given array.  It is assumed that the given
       array
4   * is sorted.  This method returns -1 if the array does not
5   * contain the element
6   */
7  public static int binarySearchRec(int a[], int low, int high, int
       element)
8  {
9    if(low > high)
```

```
10      return -1;
11    int middle_index = (high + low) / 2;  //see note
12    if(a[middle_index] == element)
13      return middle_index;
14    else if(a[middle_index] < element)
15      return binarySearchRec(a, middle_index+1, high, element);
16    else if(a[middle_index] > element)
17      return binarySearchRec(a, low, middle_index-1, element);
18  }
```

Code Sample 3: Java Iterative Binary Search for integers

```
1  /**
2   * This function returns an index at which the given element
3   * appears in the given array.  It is assumed that the given
        array
4   * is sorted.  This method returns -1 if the array does not
5   * contain the element
6   */
7  public static int binarySearch(int a[], int element)
8  {
9    int l = 0, h = a.length - 1;
10   while(l <= h) {
11     int m = (l + h) / 2;  //see note
12     if(a[m] == element)
13       return m;
14     else if(a[m] < element)
15       l = m + 1;
16     else
17       h = m - 1;
18   }
19   return -1;
20 }
```

5.2.1. Preventing Arithmetic Errors

The lines of code that compute the new mid-index, such as

```
int middle_index = (high + low)/ 2;
```

are prone to arithmetic errors in certain situations in which you are dealing with very large arrays. In particular, if the variables `high` and `low` have a sum that exceeds the maximum value that a signed 32-bit integer can hold ($2^31 - 1 = 2,147,483,647$), then overflow will occur before the division by 2, leading to a (potentially) negative number.

One solution would be to use a `long` integer instead which will prevent overflow as it would be able to handle any size array (which are limited to 32-bit signed integers (actually slightly less when considering the small amount of memory overhead for an object).

Another solution is to use operations that do not introduce this overflow. For example,

$$\frac{l+h}{2} = l + \frac{(h-l)}{2}$$

but the right hand side will not be prone to overflow. Thus the code,

```
int middle_index = low + (high - low)/ 2;
```

would work.

In fact, this bug is quite common [5] and was in the Java implementation, unreported for nearly a decade [1].

5.3. Searching in Java: Doing It Right

In practice, we don't reinvent the wheel: we don't write a custom search function for every type and for every sorting order.

Instead we:

1. Use standard search functions provided by the language
2. *Configure* using a `Comparator` rather than *Code*

- Built-in functions require that the `equals()` and `hashCode()` methods are properly overridden (See Appendix A)
- Linear search: `List.indexOf(Object o)`
- Binary Search: `int Collections.binarySearch(List list, Object key)`
  - Searches the specified list for the specified object using the binary search algorithm
  - Returns the index at which `key` appears
  - Returns something negative if not found
  - Requires that the list contains elements that have a *Natural Ordering*
- Binary Search: `int Collections.binarySearch(List, Object, Comparator)`
  - Searches the given list for the specified object using the binary search algorithm
  - Uses the provided `Comparator` to determine order (*not* the `equals()` method), see Appendix B
- Binary Search with arrays: `Arrays.binarySearch(T[] a, T key, Comparator)`

5.4. Examples

Code Sample 4: Java Search Examples

```
1 ArrayList<Student> roster = ...
2
3 Student castroKey = null;
4 int castroIndex;
```

```
5
6  //create a "key" that will match according to the Student.equals
       () method
7  castroKey = new Student("Starlin", "Castro", 131313, 3.95);
8  castroIndex = roster.indexOf(castroKey);
9  System.out.println("at index " + castroIndex + ": " + roster.get(
       castroIndex));
10
11 //create a key with only the necessary fields to match the
       comparator
12 castroKey = new Student("Starlin", "Castro", 0, 0.0);
13 //sort the list according to the comparator
14 Collections.sort(roster, byName);
15 castroIndex = Collections.binarySearch(roster, castroKey, byName)
       ;
16 System.out.println("at index " + castroIndex + ": " + roster.get(
       castroIndex));
17
18 //create a key with only the necessary fields to match the
       comparator
19 castroKey = new Student(null, null, 131313, 0.0);
20 //sort the list according to the comparator
21 Collections.sort(roster, byNUID);
22 castroIndex = Collections.binarySearch(roster, castroKey, byNUID)
       ;
23 System.out.println("at index " + castroIndex + ": " + roster.get(
       castroIndex));
```

6. Searching in C

6.1. Linear Search

Code Sample 5: C Linear Search for integers

```
1  /**
2   * This function returns the index at which the given element
        first
3   * appears in the given array.  Returns -1 if the array does not
4   * contain the element.
5   */
6  int linearSearchInt(int *a, int size, int key) {
7    int i;
8    for(i=0; i<size; i++) {
9      if(a[i] == key) {
10       return i;
11     }
```

```
12    }
13    return -1;
14  }
```

6.2. Binary Search

Code Sample 6: C Recursive Binary Search for integers

```
1  /**
2   * This function returns an index at which the given element
3   * appears in the given array.  It is assumed that the given
        array
4   * is sorted.  This method returns -1 if the array does not
5   * contain the element.
6   */
7  int binarySearchRecursive(int *a, int low, int high, int element)
8  {
9    if(low > high)
10     return -1;
11   int middle_index = low + (high - low) / 2;
12   if(a[middle_index] == element)
13     return middle_index;
14   else if(a[middle_index] < element)
15     return binarySearchRecursive(a, middle_index+1, high, element
         );
16   else if(a[middle_index] > element)
17     return binarySearchRecursive(a, low, middle_index-1, element)
         ;
18 }
```

Code Sample 7: C Iterative Binary Search for integers

```
1  /**
2   * This function returns an index at which the given element
3   * appears in the given array.  It is assumed that the given
        array
4   * is sorted.  This method returns -1 if the array does not
5   * contain the element
6   */
7  int binarySearch(int *a, int size, int element)
8  {
9    int l = 0, h = size-1;
10   while(l <= h) {
11     int m = l + (l - h) / 2;
12     if(a[m] == element)
13       return m;
14     else if(a[m] < element)
```

```
15        l = m + 1;
16      else
17        h = m - 1;
18    }
19    return -1;
20  }
```

6.3. Searching in C: Doing it Right

In practice, we don't reinvent the wheel: we don't write a custom search function for every type and for every sorting order.

Instead we:

1. Use standard search functions provided by the language
2. *Configure* using a *comparator function* rather than *Code* (see Appendix D)

6.3.1. Linear Search

The C library `search.h` provides a linear search function to search arrays.

```
void *lfind(const void *key, const void *base, size_t *nmemb, size_t size, int(*
compar)(const void *, const void *));
```

- `key` - a dummy struct matching what youre looking for (can build an instance with only the required fields)
- `base` - a pointer to the array to be searched
- `nmemb` - the size of the array (number of members)
- `size` - size of each element (use `sizeof()`)
- `compar` - a pointer to a comparator function (see Appendices)
- Returns a pointer to the first instance matching the key, returns `NULL` if not found

6.3.2. Binary Search

The C standard library (`stdlib.h`) provides a binary search function to search arrays.

```
void *bsearch(const void *key, const void *base, size_t nmemb, size_t size, int
 (*compar)(const void *, const void *));
```

- `key` - a dummy struct matching what you are looking for (can build an instance with only the required fields)
- `base` - a pointer to the array to be searched
- `nmemb` - the size of the array (number of members)
- `size` - size of each element (use `sizeof()`)
- `compar` - a pointer to a comparator function (see Appendices)

- Returns the first element that it finds (not necessarily the first in the order)
- Returns `NULL` if not found
- *Assumes* the array is sorted according to the same comparator function, otherwise undefined behavior

6.4. Examples

Code Sample 8: C Search Examples

```c
#include<stdio.h>
#include<stdlib.h>
#include<search.h>

#include "student.h"

int main(int argc, char **argv) {

  int n = 0;
  Student *roster = loadStudents("student.data", &n);
  int i;
  size_t numElems = n;


  printf("Roster: \n");
  printStudents(roster, n);

  /* Searching */
  Student *castro = NULL;
  Student *castroKey = NULL;
  Student *sandberg = NULL;
  char *str = NULL;

  castro = linearSearchStudentByNuid(roster, 10, 131313);
  str = studentToString(castro);
  printf("castro: %s\n", str);
  free(str);

  //create a key that will match according to the NUID
  int nuid = 23232323;
  Student * key = createEmptyStudent();
  key->nuid = nuid;

  //use lfind to find the first such instance:
  //sandberg =
  sandberg = lfind(key, roster, &numElems, sizeof(Student),
     studentIdCmp);
```

```
37    str = studentToString(sandberg);
38    printf("sandberg: %s\n", str);
39    free(str);
40
41    //create a key with only the necessary fields
42    castroKey = createStudent("Starlin", "Castro", 0, 0.0);
43    //sort according to a comparator function
44    qsort(roster, n, sizeof(Student), studentLastNameCmp);
45
46    castro = bsearch(castroKey, roster, n, sizeof(Student),
          studentLastNameCmp);
47    str = studentToString(castro);
48    printf("castro (via binary search): %s\n", str);
49    free(str);
50
51    //create a key with only the necessary fields
52    castroKey = createStudent(NULL, NULL, 131313, 0.0);
53    //sort according to a comparator function
54    qsort(roster, n, sizeof(Student), studentIdCmp);
55
56    castro = bsearch(castroKey, roster, n, sizeof(Student),
          studentIdCmp);
57    str = studentToString(castro);
58    printf("castro (via binary search): %s\n", str);
59    free(str);
60
61    return 0;
62 }
```

Part II.
Sorting

Problem 2 (Sorting).
Given: *a collection of* orderable *elements,* $A = \{a_1, a_2, \ldots, a_n\}$
Output: *A permuted list of elements* $A' = \{a'_1, a'_2, \ldots, a'_n\}$ *according to a specified order*

- Simple, but ubiquitous problem
- Fundamental operation for data processing
- Large variety of algorithms, data structures, applications, etc.

Sorting algorithms are usually analyzed with respect to:

- Number of comparisons in the worst, best, average cases
- Swaps

- Extra memory required

Other considerations

- Practical considerations: what ordered collections (Lists, arrays, etc.) are supported by a language
- Most languages provide standard (optimized, well-tested) sorting functionality; *use it!*

Sorting stability: A sorting algorithm is said to be *stable* if the relative order of "equal" elements is preserved.

- Example: suppose that a list contains $10, 2_a, 5, 2_b$; a stable sorting algorithm would produce $2_a, 2_b, 5, 10$ while a non-stable sorting algorithm may produce $2_b, 2_a, 5, 10$.
- Stable sorts are important for data presentation (sorting by two columns/categories)
- Stability depends on inequalities used and behavior of algorithms

Throughout, we will demonstrate examples of sorting based on the array in Figure 2.

4	12	8	1	42	23	7	3

Figure 2: Unsorted Array

7. Bubble Sort

7.1. Basic Idea

- Pass through the list and swap individual pairs of elements if they are out of order
- At the end of the first pass, the maximal element has "bubbled up" to the end of the list
- Repeat this process n times
- Pseudocode presented in Algorithm 4
- Java code (for integers) presented in Code Sample 9
- C code (for integers) presented in Code Sample 10
- Example

7.2. Pseudocode

```
   INPUT  : A collection A = {a_1, ..., a_n}
   OUTPUT: An array A' containing all elements of A in nondecreasing order
1  FOR i = 1, ..., (n − 1) DO
2  |  FOR j = 2, ..., (n − i + 1) DO
3  |  |  IF a_{j−1} > a_j THEN
4  |  |  |  swap a_{j−1}, a_j
5  |  |  END
6  |  END
7  END
```

Algorithm 4: Bubble Sort

7.3. Analysis

- Elementary operation: comparison
- Performed once on line 3
- Inner loop (line 2 executed $n - i$ times
- Outer loop (line 1) executed n times
- In total:

$$\sum_{i=1}^{n} \sum_{j=2}^{(n-i+1)} 1 = \sum_{i=1}^{n} (n-i) = n^2 - \left(\sum_{i=1}^{n} i \right) = \frac{n^2 - n}{2}$$

- Bubble sort is $O(n^2)$

7.4. Code Samples

Code Sample 9: Java Bubble Sort for integers

```
1  public static void bubbleSort(int array[]) {
2    int n = array.length;
3    int temp = 0;
4    for(int i = 0; i < n; i++) {
5      for(int j = 1; j < (n-i); j++) {
6        if(array[j-1] > array[j]) {
7          temp = array[j-1];
8          array[j-1]=array[j];
9          array[j]=temp;
10       }
11     }
12   }
```

```
}
```

Code Sample 10: C Bubble Sort for integers

```
void bubbleSortInt(int *array, int n) {
  int temp = 0;
  for(int i = 0; i < n; i++) {
    for(int j = 1; j < (n-i); j++) {
      if(array[j-1] > array[j]) {
        temp = array[j-1];
        array[j-1]=array[j];
        array[j]=temp;
      }
    }
  }
}
```

8. Selection Sort

8.1. Basic Idea

- Iterate through the elements in the list and find the minimal element in the list
- Swap the minimal element with the "first" element
- Repeat the process on the remaining $n-1$ elements
- Pseudocode presented in Algorithm 5
- Java code (for integers) presented in Code Sample 11
- C code (for integers) presented in Code Sample 12
- Example
- Note: Selection sort is not stable, an example: $2_a, 2_b, 1, 5$; the first swap would result in $1, 2_b, 2_a, 5$ and no subsequent changes would be made.

8.2. Pseudocode

```
   INPUT  : A collection A = {a_1, ..., a_n}
   OUTPUT: An array A' containing all elements of A in nondecreasing order
 1 FOR i = 1, ..., (n − 1) DO
 2 |   a_min ← a_i
 3 |   FOR j = (i + 1), ..., n DO
 4 |   |   IF a_min > a_j THEN
 5 |   |   |   min ← a_j
 6 |   |   END
 7 |   END
 8 |   swap a_min and a_i
 9 END
```

Algorithm 5: Selection Sort

8.3. Analysis

- Comparison performed: once, inner loop, outer loop
- In total:

$$\sum_{i=1}^{n-1} \sum_{j=(i+1)}^{n} 1 = \sum_{i=1}^{n-1} (n-i) = n(n-1) - \sum_{i=1}^{n-1} i = n(n-1) - \frac{n(n-1)}{2} = \frac{n(n-1)}{2}$$

- Selection Sort is $O(n^2)$

8.4. Code Samples

Code Sample 11: Java Selection Sort for integers

```
1  public static void selectionSort(int array[]) {
2    int n = array.length;
3    for(int i=0; i<n; i++){
4      int index_of_min = i;
5      for(int j=i; j<n; j++){
6        if(array[index_of_min] > array[j]){
7          index_of_min = j;
8        }
9      }
10     int temp = array[i];
11     array[i] = array[index_of_min];
12     array[index_of_min] = temp;
13   }
```

```
14 }
```

Code Sample 12: C Selection Sort for integers

```
1  void selectionSort(int *array, int n) {
2    for(int i=0; i<n; i++){
3      int index_of_min = i;
4      for(int j=i; j<n; j++){
5        if(array[index_of_min] > array[j]){
6          index_of_min = j;
7        }
8      }
9      int temp = array[i];
10     array[i] = array[index_of_min];
11     array[index_of_min] = temp;
12   }
13 }
```

9. Insertion Sort

9.1. Basic Idea

- If we consider just the first element, the list is *sorted*
- Now consdier the second element: we can *insert* it in the currently sorted sublist
- In general: at the k-th iteration, assume that elements $a_1, \ldots, a_k$ are sorted
- Insert element a_{k+1} into the proper location among elements $a_1, \ldots, a_k$
- Insertion is performed by *shifting* a_{k+1} down until its in the right spot
- Pseudocode presented in Algorithm 6
- Java code (for integers) presented in Code Sample 13
- C code (for integers) presented in Code Sample 14
- Example

9.2. Pseudocode

```
   INPUT  : A collection A = {a_1, ..., a_n}
   OUTPUT: An array A' containing all elements of A in nondecreasing order
1  FOR i = 2, ..., n DO
2  |  x ← a_i
3  |  j ← i
4  |  WHILE j > 1 and a_{j-1} > x DO
5  |  |  a_j ← a_{j-1}
6  |  |  decrement j
7  |  END
8  |  a_j ← x
9  END
```

Algorithm 6: Insertion Sort

9.3. Analysis

- Best case: list is already sorted, number of comparisons is $(n-1)$
- Worst case: reversed list; each iteration shifts the element all the way down
- First iteration: 1 comparison, 2nd: at most 2, etc., last iteration: at most $(n-1)$ comparisons

$$\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

- Worst case: $O(n^2)$
- A more complex analysis for the *average case*, but still $O(n^2)$
- Practical consideration: insertion sort is inherently adaptive; good performance in practice
- Very efficient on small lists, especially if there is already some structure (partially ordered) to them

9.4. Code Samples

Code Sample 13: Java Insertion Sort for integers

```
1 public static void insertionSort(int array[]) {
2   int value;
3   for(int i=1; i<array.length; i++) {
4     value = array[i];
5     int j = i;
```

```
6       while(j > 0 && array[j-1] > value) {
7         array[j] = array[j-1];
8         j--;
9       }
10      array[j] = value;
11    }
12  }
```

Code Sample 14: C Insertion Sort for integers

```
1  void insertionSort(int *array, int size) {
2    int value;
3    for(int i=1; i<size; i++) {
4      value = array[i];
5      int j = i;
6      while(j > 0 && array[j-1] > value) {
7        array[j] = array[j-1];
8        j--;
9      }
10     array[j] = value;
11   }
12 }
```

10. Quick Sort

Due to Tony Hoare in the early 1960s [2, 3]

10.1. Basic Idea

- Choose a *pivot* element a_p
- Partition all other elements *around* this pivot: move all less elements to the "left" side of the list; all greater elements to the "right"
- Place pivot element between them
- Repeat the process on the two partitions until each sublist is trivially sorted (empty or a single element)
- Pseudocode presented in Algorithm 7 and 8
- Java code (for integers) presented in Code Sample 15
- C code (for integers) presented in Code Sample 17
- Example

10.2. Pseudocode

```
   INPUT  : A collection A = {a_1, ..., a_n}, indices l, r
   OUTPUT: An array A' containing all elements of A in nondecreasing order
1  IF l < r THEN
2  |  s = PARTITION(A[l ... r])
3  |  QUICKSORT(A[l ... (s − 1)])
4  |  QUICKSORT(A[(s + 1) ... r])
5  END
```

Algorithm 7: Quick Sort

```
   INPUT  : Integer array A, sub-indices 1 ≤ l, r ≤ n
   OUTPUT: A partitioned integer array A' such that all
           A[l ... (j − 1)] ≤ A[j] ≤ A[(j + 1) ... r] and an index j of the pivot
           element.
1  pivot ← A[l]
2  i ← l
3  j ← (r + 1)
4  REPEAT
5  |  REPEAT
6  |  |  i ← (i + 1)
7  |  UNTIL a_i ≥ pivot
8  |  REPEAT
9  |  |  j ← (j − 1)
10 |  UNTIL a_j ≤ pivot
11 |  swap a_i, a_j
12 UNTIL i ≥ j
13 swap a_i, a_j //Undo the last swap
14 swap a_l, a_j //Swap the pivot
15 return j
```

Algorithm 8: In-Place Partition

10.3. Analysis

- Performance of quick sort is highly dependent on how the *even* the partitioning is
- Partitioning depends on pivot choice

Best Case

- Ideal pivot choice: always choose the median element

- Evenly divides the list into two equal parts
- Two calls to quick sort on lists roughly half the size
- A linear number of comparisons to partition
- Recurrence:
$$C(n) = 2C\left(\frac{n}{2}\right) + n$$
- By the Master Theorem, $O(n \log n)$

Worst Case

- Worst pivot choice: an extremal element
- Divides into an empty list and a list of size $(n-1)$
- Only one recursive call is necessary, but on a list of size $(n-1)$
- Recurrence:
$$C(n) = C(n-1) + n$$
- Back substitution yields $O(n^2)$

Average Case

- Even if the list is is split by a constant proportion, still $O(n \log n)$
- A careful average case analysis also yields
$$C(n) \approx 1.38n \log n$$
- Still $O(n \log n)$

Pivot Choice: Pivot choice greatly affects performance; several strategies:

- Median-of-Three – Among three elements choose the median.
  - Guarantees that the pivot choice will never be the worst case.
  - Does not guarantee $\Theta(n \log n)$.
- Random Element – Randomly select an index for a pivot element.
  - Guarantees *average* running time of $\Theta(n \log n)$.
  - Extra work to randomly select a pivot.
- Linear Time Median Finding.
  - An algorithm exists that runs in $\Theta(n)$ time to find the median of n objects.
  - Guarantees $\Theta(n \log n)$ in all cases.
  - Complicated; recursive; *huge* overhead.

10.4. Code Samples

Code Sample 15: Java Quick Sort for integers

```
1 private static void quickSort(int a[], int left, int right) {
```

```
  int index = partition(a, left, right);
  if (left < index - 1)
    quickSort(a, left, index - 1);
  if (index < right)
    quickSort(a, index, right);
}
```

Code Sample 16: Java Partition subroutine for integers

```
public static int partition(int a[], int left, int right)
{
  int i = left, j = right;
  int tmp;
  int pivot = a[(left + right) / 2];

  while (i <= j) {
    while (a[i] < pivot)
      i++;
    while (a[j] > pivot)
      j--;
    if (i <= j) {
      tmp = a[i];
      a[i] = a[j];
      a[j] = tmp;
      i++;
      j--;
    }
  }
  return i;
}
```

Code Sample 17: C Quick Sort for integers

```
void swap(int *a, int *b)
{
  int t=*a; *a=*b; *b=t;
}
void sort(int arr[], int beg, int end)
{
  if (end > beg + 1)
  {
    int piv = arr[beg], l = beg + 1, r = end;
    while (l < r)
    {
      if (arr[l] <= piv)
        l++;
      else
        swap(&arr[l], &arr[--r]);
```

```
16      }
17      swap(&arr[--l], &arr[beg]);
18      sort(arr, beg, l);
19      sort(arr, r, end);
20    }
21  }
```

A good tutorial with a non-recursive C implementation: http://alienryderflex.com/quicksort/

11. Merge Sort

Due to John von Neumann, 1945 (as reported by Knuth [4]).

11.1. Basic Idea

- Divide the list into two (roughly) equal parts and recursively sort each
- Recursion stops when the list is trivially sorted (empty or a single element)
- When returning from the recursion, *merge* the two lists together
- Since both sublists are sorted, we only need to maintain an index to each list, add the least element, etc.
- Only a linear amount of work to merge two sorted lists
- Pseudocode presented in Algorithm 9
- Java code (for integers) presented in Code Sample 18
- C code (for integers) presented in Code Sample 20
- Example

11.2. Pseudocode

INPUT : An array, sub-indices $1 \leq l, r \leq n$
OUTPUT: An array A' such that $A[l, \ldots, r]$ is sorted

1 IF $l < r$ THEN
2 | MERGESORT$(A, l, \lfloor \frac{r+l}{2} \rfloor)$
3 | MERGESORT$(A, \lceil \frac{r+l}{2} \rceil, r)$
4 | Merge sorted lists $A[l, \ldots, \lfloor \frac{r+l}{2} \rfloor]$ and$A[\lceil \frac{r+l}{2} \rceil, \ldots, r]$
5 END

Algorithm 9: Merge Sort

11.3. Analysis

- Because merge sort divides the list *first*, an even split is *guaranteed*
- After the recursion, a linear amount of work is done to merge the two lists
- Basic idea:
  - Keep two index pointers to each sublist
  - Copy the minimal element, advance the pointer
  - Until one list is empty, then just copy the rest of the other
- Requires use of $O(n)$-sized temporary array
- Recurrence:

$$C(n) = 2C\left(\frac{n}{2}\right) + n$$

- By the Master Theorem, $O(n \log n)$
- In fact, a guarantee (best, worst, average are the same)

11.4. Code Samples

Code Sample 18: Java Merge Sort for integers

```
1 public static void mergeSort(int a[], int low, int high)
2 {
3   if (low < high) {
4     int middle = (low + high) / 2;
5     mergesort(low, middle);
6     mergesort(middle + 1, high);
7     merge(low, middle, high);
8   }
9 }
```

Code Sample 19: Java Merge for integers

```
1  private static int MERGE_HELPER[];
2  ...
3  private static void merge(int a[], int low, int middle, int high)
       {
4
5    // Copy both parts into the helper array
6    for (int i = low; i <= high; i++) {
7      MERGE_HELPER[i] = a[i];
8    }
9    int i = low;
10   int j = middle + 1;
11   int k = low;
```

```
12    //Copy the smallest values from the left or the right side back
          to the original array
13    while (i <= middle && j <= high) {
14      if (MERGE_HELPER[i] <= MERGE_HELPER[j]) {
15        a[k] = MERGE_HELPER[i];
16        i++;
17      } else {
18        a[k] = MERGE_HELPER[j];
19        j++;
20      }
21      k++;
22    }
23    // Copy the rest of the left side of the array into the target
          array
24    while (i <= middle) {
25      a[k] = MERGE_HELPER[i];
26      k++;
27      i++;
28    }
29    // Copy the rest of the right side of the array into the target
          array
30    while (j <= high) {
31      a[k] = MERGE_HELPER[j];
32      k++;
33      j++;
34    }
35  }
```

Code Sample 20: C Merge Sort for integers

```
1  void mergeSort(int *array, int left, int right)
2  {
3    int mid = (left+right)/2;
4    /* We have to sort only when left<right because when left=right
         it is anyhow sorted*/
5    if(left<right)
6    {
7      /* Sort the left part */
8      mergeSort(array,left,mid);
9      /* Sort the right part */
10     mergeSort(array,mid+1,right);
11     /* Merge the two sorted parts */
12     merge(array,left,mid,right);
13     }
14 }
```

The function `mergeSort` would be called as follows:

```
int *a = (int *) malloc(sizeof(int) * n);
...
mergeSort(a, 0, n-1);
```

Code Sample 21: C Merge Sort for integers

```
void merge(int *array, int left, int mid, int right)
{
  /* temporary array to store the new sorted part */
  int tempArray[right-left+1];
  int pos=0, lpos = left, rpos = mid + 1; //not strict ANSI C
  while(lpos <= mid && rpos <= right) {
    if(array[lpos] < array[rpos]) {
      tempArray[pos++] = array[lpos++];
    } else {
      tempArray[pos++] = array[rpos++];
    }
  }
  while(lpos <= mid)  tempArray[pos++] = array[lpos++];
  while(rpos <= right)tempArray[pos++] = array[rpos++];
  int iter;
  /* copy back the sorted array to the original array */
  for(iter=0; iter < pos; iter++) {
    array[iter+left] = tempArray[iter];
  }
  return;
}
```

12. Heap Sort

Due to J. W. J. Williams, 1964 [7].

See Tree Note Set

13. Tim Sort

Due to Tim Peters in 2002 [6] who originally wrote it for the Python language. It has since been adopted in Java (version 7 for arrays of non-primitive types).

TODO

Algorithm	Complexity			Stability	Notes
	Best	Average	Worst		
Bubble Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Stable	
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Not Stable	
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	Stable	Best in practice for small collections
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	Not Stable	Performance depends on pivot choices
Merge Sort	$\Theta(n \log n)$	$O(n \log n)$	$O(n \log n)$	Stable	Guaranteed $\Theta(n \log n)$

Table 3: Summary of Sorting Algorithms

14. Summary

15. In Practice: Sorting in Java

Arrays:

- `java.util.Arrays` provides a `sort` method for all primitive types in ascending order
- `sort(Object[] a)` allows you to sort arrays of objects that have a *natural ordering*: classes that implement the `Comparable` interface
- `sort(T[] a, Comparator<? super T> c)` allows you to sort objects according to the order defined by a provided `Comparator` (see Appendix B)

Lists:

- `java.util.Collections` provides two sort methods to sort `List` collections
- `sort(List<T> list)` – sorts in ascending order according to the *natural ordering*
- `sort(List<T> list, Comparator<? super T> c)` – sorts according to the order defined by the given comparator
- Java specification dictates that the sorting algorithm *must* be stable
- Java 1 – 6: hybrid merge/insertion sort
- Java 7: "timsort" (a bottom-up merge sort that merges "runs" of ordered sub lists)

15.1. Considerations

When sorting collections or arrays of objects, we may need to consider the possibility of uninitialized `null` objects. How we handle these are a design decision. We could ignore it in which case such elements would likely result in a `NullPointerException` and expect the user to prevent or handle such instances. This may be the preferable choice in most

instances, in fact.

Alternatively, we could handle `null` objects in the design of our Comparator. Code Snippet 22 presents a comparator for our `Student` class that orders `null` instances first.

Code Sample 22: Handling Null Values in Java Comparators

```
Comparator<Student> byNameWithNulls = new Comparator<Student>()
    {
  @Override
  public int compare(Student a, Student b) {
    if(a == null && b == null) {
      return 0;
    } else if(a == null && b != null) {
      return -1;
    } else if(a != null && b == null) {
      return 1;
    } else {
      if(a.getLastName().equals(b.getLastName())) {
        return a.getFirstName().compareTo(b.getFirstName());
      } else {
        return a.getLastName().compareTo(b.getLastName());
      }
    }
  }
};
```

15.2. Examples

Code Sample 23: Using Java Collection's Sort Method

```
List<Student> roster = ...
Student rosterArr[] = ...
Comparator byName = ...
Comparator byGPA = ...

//sort by name:
Collections.sort(roster, byName);
Arrays.sort(rosterArr, byName);

//sort by GPA:
Collections.sort(roster, byGPA);
Arrays.sort(rosterArr, byGPA);
```

16. In Practice: Sorting in C

The standard C library (`stdlib.h`) provides a Quick sort implementation:

```
void qsort(void *base, size_t nmemb, size_t size, int(*compar)(const void *, const
 void *))
```

- `base` – pointer an array of elements
- `nmemb` – the size of the array (number of members)
- `size` – the size (in bytes) of each element (use `sizeof`)
- `compar` – a comparator function used to order elements
- Sorts in ascending order according to the provided comparator function

Advantages:

- No need to write a new sorting algorithm for every user defined type and every possible order along every possible component
- Only need to create a simple comparator function
- Less code, less chance of bugs
- `qsort` is well-designed, highly optimized, well tested, proven
- Prefer *configuration* over coding
- Represents a weak form of polymorphic behavior (same code can be executed on different types)

16.1. Sorting Pointers to Elements

Comparator functions take void pointers to the elements that they are comparing. Often, you have need to sort an array of *pointers* to elements. The most common use case for this is using `qsort` to sort strings.

An array of strings can be thought of as a 2-dimensional array of `char`s. Specifically, an array of strings is a `char **` type. That is, an array of pointers to `char`s. We may be tempted to use `strcmp` in the standard string library, passing it to `qsort`. Unfortunately this will not work. `qsort` requires two `const void *` types, while `strcmp` takes two `const char *` types. This difference is subtle but important; a full discussion can be found on the c-faq (http://c-faq.com/lib/qsort1.html).

The recommended way of doing this is to define a different comparator function as follows.

Code Sample 24: C Comparator Function for Strings

```
1 /* compare strings via pointers */
2 int pstrcmp(const void *p1, const void *p2)
3 {
4   return strcmp(*(char * const *)p1, *(char * const *)p2);
5 }
```

Observe the behavior of this function: it uses the standard `strcmp` function, but makes the proper explicit type casting before doing so. The `*(char * const *)` casts the generic void pointers as pointers to strings (or pointers to pointers to characters), then dereferences it to be compatible with `strcmp`.

Another case is when we wish to sort user defined types. The `Student` structure presented earlier is "small" in that it only has a few fields. When structures are stored in an array and sorted, there may be many *swaps* of individual elements which involves a lot of memory copying. If the structures are small this is not too bad, but for "larger" structures this could be potentially expensive. Instead, it may be preferred to have an array of *pointers* to structures. Swapping elements involves only swapping pointers instead of the entire structure. This is far cheaper as a memory address is likely to be far smaller than the actual structure it points to. This is essentially equivalent to the string scenario: we have an array of pointers to be sorted, our comparator function then needs to deal with pointers to pointers. A full discussion can be found on c-faq (http://c-faq.com/lib/qsort2.html). An example appears in Code Snippet 25.

Code Sample 25: Sorting Structures via Pointers

```
1   //An array of pointers to Students
2   Student **roster = (Student **) malloc(sizeof(Student *) * n);
3   qsort(roster, n, sizeof(Student *), studentPtrLastNameCmp);
4
5   ...
6
7  int studentPtrLastNameCmp(const void *s1, const void *s2) {
8   //we receive a pointer to an individual element in the array
9   // but individual elements are POINTERS to students
10  // thus we cast them as (const Student **)
11  // then dereference to get a poitner to a student!
12  const Student *a = *(const Student **)s1;
13  const Student *b = *(const Student **)s2;
14  int result = strcmp(a->lastName, b->lastName);
15  if(result == 0) {
16    return strcmp(a->firstName, b->firstName);
17  } else {
18    return result;
19  }
20 }
```

Another issue when sorting arrays of pointers is that we may now have to deal with `NULL` elements. When sorting arrays of elements this is not an issue as a properly initialized array will contain non-null elements (though elements could still be uninitialized, the memory space is still valid).

How we handle `NULL` pointers is more of a design decision. We could ignore it and any attempt to access a `NULL` structure will result in undefined behavior (or segmentation faults, etc.). Or we could give `NULL` values an explicit ordering with respect to other elements. That is, we could order all `NULL` pointers *before* non-`NULL` elements (and consider

all `NULL` pointers to be equal). An example with respect to our `Student` structure is given in Code Snippet 26.

Code Sample 26: Handling Null Values

```
int studentPtrLastNameCmpWithNulls(const void *s1, const void *s2
    ) {
  const Student *a = *(const Student **)s1;
  const Student *b = *(const Student **)s2;
  if(a == NULL && b == NULL) {
    return 0;
  } else if(a == NULL && b != NULL) {
    return -1;
  } else if (a != NULL && b == NULL) {
    return 1;
  }
  int result = strcmp(a->lastName, b->lastName);
  if(result == 0) {
    return strcmp(a->firstName, b->firstName);
  } else {
    return result;
  }
}
```

16.2. Examples

Code Sample 27: Using C's `qsort` function

```
#include<stdio.h>
#include<stdlib.h>

#include "student.h"


int main(int argc, char **argv) {

  int n = 0;
  Student *roster = loadStudents("student.data", &n);
  int i;
  size_t numElems = n;

  printf("Roster: \n");
  printStudents(roster, n);

  printf("\n\n\nSorted by last name/first name: \n");
  qsort(roster, numElems, sizeof(Student), studentLastNameCmp);
  printStudents(roster, n);
```

```
20
21    printf("\n\n\nSorted by ID: \n");
22    qsort(roster, numElems, sizeof(Student), studentIdCmp);
23    printStudents(roster, n);
24
25    printf("\n\n\nSorted by ID, descending: \n");
26    qsort(roster, numElems, sizeof(Student), studentIdCmpDesc);
27    printStudents(roster, n);
28
29    printf("\n\n\nSorted by GPA: \n");
30    qsort(roster, numElems, sizeof(Student), studentGPACmp);
31    printStudents(roster, n);
32
33    return 0;
34 }
```

A. Java: `equals` and `hashCode` methods

- Every class in Java is a sub-class of the `java.lang.Object` class
- `Object` defines several methods whose default behavior is to operate on Java Virtual Machine memory addresses:
  - `public String toString()` – returns a hexadecimal representation of the memory address of the object
  - `public int hashCode()` – may convert the memory address to an integer and return it
  - `public boolean equals(Object obj)` – will compare `obj` to `this` instance and return `true` or `false` if they have the same or different memory address
- When defining your own objects, its best practice to override these methods to be dependent on the entire state of the object
- Doing so is *necessary* if you use your objects in the Collections library
- If not, then creating a key object will never match an array element as `equals` will always return false (since the key and any actual element will not have the same memory address)
- Good tutorial: http://www.javapractices.com/topic/TopicAction.do?Id=17
- Eclipse Tip: Let Eclipse do it for you! Source → Generate equals and hashCode, Generate toString, etc.

A.1. Example

Code Sample 28: Java `Student` class

```java
package unl.cse.searching_sorting;

public class Student {

  private String firstName;
  private String lastName;
  private Integer nuid;
  private Double gpa;

  public Student(String firstName, String lastName, int nuid,
      double gpa) {
    this.firstName = firstName;
    this.lastName = lastName;
    this.nuid = nuid;
    this.gpa = gpa;
  }

  @Override
  public String toString() {
    StringBuilder sb = new StringBuilder();
    sb.append(lastName)
      .append(", ")
      .append(firstName)
      .append(" (")
      .append(nuid)
      .append("), ")
      .append(gpa);
    return sb.toString();
  }

  @Override
  public boolean equals(Object obj) {
    if (this == obj)
      return true;
    if (obj == null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    Student that = (Student) obj;
    boolean equal = true;
    equal = equal && (this.firstName != null ? this.firstName.
       equals(that.firstName) : that.firstName == null);
    equal = equal && (this.lastName != null ? this.lastName.
       equals(that.lastName) : that.lastName == null);
    equal = equal && (this.nuid != null ? this.nuid.equals(that.
       nuid) : that.nuid == null);
```

```
43      equal = equal && (this.gpa != null ? this.gpa.equals(that.gpa
            ) : that.gpa == null);
44      return equal;
45    }
46
47    @Override
48    public int hashCode() {
49      int hash = 7;
50      hash = (31 * hash) + nuid.hashCode();
51      hash = (31 * hash) + gpa.hashCode();
52      hash = (31 * hash) + firstName.hashCode();
53      hash = (31 * hash) + lastName.hashCode();
54      return hash;
55    }
56
57  }
```

B. Java: The `Comparator` Interface

- Resource:
  http://docs.oracle.com/javase/tutorial/collections/interfaces/order.html
- Interface documentation:
  http://docs.oracle.com/javase/6/docs/api/java/util/Comparator.html
- Objects may have a natural ordering (built-in types or objects that are `Comparable`), otherwise `Comparator` classes may be created that define an ordering
- `Comparator<T>` is a parameterized interface
  - `T` is the type that the comparator is used on (`Integer, Double, Student`)
  - As an interface, it only specifies one method:
    `public int compare(T a, T b)`
  - Basic contract: returns
    - Something negative if $a < b$
    - Zero if a equals b
    - Something positive if $a > b$
- Usual to create comparators as anonymous classes (classes created and defined inline, not in a separate class; comparators are ad-hoc, use-as-needed classes)
- Design Tip: Don't Repeat Yourself; utilize comparison functions already provided by the language!
- Take care with algebraic tricks (subtraction) to return a difference:
  - Some combinations may not give correct results due to overflow
  - Differences with floating point numbers may give incorrect results when trun-

cated to integers

B.1. Examples

Code Sample 29: Java Comparator Examples for the `Student` class

```
Comparator<Student> byName = new Comparator<Student>() {
  @Override
  public int compare(Student a, Student b) {
    if(a.getLastName().equals(b.getLastName())) {
      return a.getFirstName().compareTo(b.getFirstName());
    } else {
      return a.getLastName().compareTo(b.getLastName());
    }
  }
};

Comparator<Student> byNameDesc = new Comparator<Student>() {
  @Override
  public int compare(Student a, Student b) {
    if(b.getLastName().equals(a.getLastName())) {
      return b.getFirstName().compareTo(a.getFirstName());
    } else {
      return b.getLastName().compareTo(a.getLastName());
    }
  }
};

Comparator<Student> byNUID = new Comparator<Student>() {
  @Override
  public int compare(Student a, Student b) {
    return (a.getNUID() - b.getNUID());
  }
};
```

B.2. Exercises

1. Write comparator instances to order `Student` objects by GPA, NUID descending, NUID as a padded out `string`
2. Write a custom sorting algorithm that uses a comparator to sort
3. Consider what happens with these comparators if we attempt to compare a `null` object with another object. How can we implement these differently?

C. C: Function Pointers

- Pointers (references) point to memory locations
- Pointers can point to: simple data types (int, double), user types (structs), arrays, etc.
- Pointers can be generic, called *void pointers*: `(void *)`
- Void pointers just point to the start of a memory block, not a specific type
- A program's code also lives in memory; each function's code lives somewhere in memory
- Therefore: pointers can also point to functions (the memory address where the code for the function is stored)!
- The pointer to a function is simply its identifier (function name)
- Usage: in callbacks: gives the ability to specify a function that another function should call/use (Graphical User Interface and Event Driven Programming)
- Declaration syntax:
  `int (*ptrToFunc)(int, double, char)= NULL;`
  (`ptrToFunc` is a pointer to a function that takes three parameters (an `int`, `double`, and `char`) and returns an `int`
- Components:
  - Return type
  - Name of the pointer
  - Parameter list type
- Assignment Syntax:
  `ptrToFunc = functionName;` or
  `ptrToFunc = &functionName;`
- Function pointers can be used as function parameters
- Functions can also be called (invoked) using a function pointer
- Passing as an argument to a function: just pass the function name!
- Function Pointers in C tutorial: http://www.newty.de/fpt/index.html

Note: some languages treat functions as "first-class citizens", meaning that variables can be numeric, strings, etc., but also functions! In C, such functionality is achieved through the use of function pointers.

C.1. Full Example

Code Sample 30: C Function Pointer Examples

```
1 #include<stdio.h>
2 #include<stdlib.h>
```

```
3
4  int function01(int a, double b);
5  void function02(double x, char y);
6
7  void runAFunction(int (*theFunc)(int, double));
8
9  int main(int argc, char **arg) {
10
11   int i = 5;
12   double d = 3.14;
13   char c = 'Q';
14
15   //calling a function normally...
16   int j = function01(i, d);
17   function02(d, c);
18
19   //function pointer declaration
20   int (*pt2Func01)(int, double) = NULL;
21   void (*pt2Func02)(double, char) = NULL;
22
23   //assignment
24   pt2Func01 = function01;
25   //or:
26   pt2Func01 = &function01;
27   pt2Func02 = &function02;
28
29   //you can invoke a function using a pointer to it:
30   j = pt2Func01(i, d);
31   pt2Func02(d, c);
32
33   //alternatively, you can invoke a function by dereferencing it:
34   j = (*pt2Func01)(i, d);
35   (*pt2Func02)(d, c);
36
37   //With function pointers, you can now pass entire functions as
        arguments to another function!
38   printf("Calling runAFunction...\n");
39   runAFunction(pt2Func01);
40   //we should not pass in the second pointer as it would not
        match the signature:
41   //syntactically okay, compiler warning, undefined behavior
42   //runAFunction(pt2Func02);
43 }
44
45 void runAFunction(int (*theFunc)(int, double)) {
46
47   printf("calling within runAfunction...\n");
```

```c
  int result = theFunc(20, .5);
  printf("the result was %d\n", result);

  return;
}

int function01(int a, double b) {
  printf("You called function01 on a = %d, b = %f\n", a, b);
  return a + 10;
}

void function02(double x, char y) {

  printf("You called function02 on x = %f, y = %c\n", x, y);

}
```

D. C: Comparator Functions

Motivation:

- Compiler "knows" how to compare built-in primitive types (you can directly use comparison operators, `<`, `>`, `<=`, `>=`, `==`, `!=`
- Need a way to search and sort for user defined types
- Sorting user-defined types (`Student`, `Album`, etc.) according to some field or combination of fields would require a specialized function for each type
- Ascending, descending, with respect to any field or combination of fields: requires yet-another-function?
- Should not have to reinvent the wheel every time—we should be able to use the same basic sorting algorithm but configured to give us the order we want
- In C: we make use of a *comparator function* and function pointers
- Utilize standard library's sort (`qsort`) and search (`bsearch`) functions

Comparator Function:

- All comparator functions have the signature:
  `int(*compar)(const void *a, const void *b)`
  - `compar` is the function name, it should be unique and descriptive
  - Arguments: two elements to be compared to each other; note: `const` and `void *`
  - Returns an integer
- Function contract: Returns

  - Something negative if `a` precedes (is "less than") `b`
  - Zero if `a` is "equal" to `b`
  - Something positive if `a` succeeds (is "greater than") `b`
- Argument types are `const`, so guaranteed not to change
- Argument types are generic void pointers: `void *` to make comparator functions as general as possible

Standard Implementation Pattern:

1. Make the general void pointers into a pointer to a specific type by making an explicit type cast:
2. Use their state (one of their components or a combination of components) to determine their order
3. Return an integer that expresses this order

Design tips:

- Make use of available comparator functions (`strcmp`, etc.)
- Reverse order: use another comparator and "flip" its value!
- Take care with algebraic tricks (subtraction) to return a difference:
  - Some combinations may not give correct results due to overflow
  - Differences with floating point numbers may give incorrect results when truncated to integers

D.1. Examples

Code Sample 31: C Structure Representing a Student

```c
/**
 * A structure to represent a student
 */
typedef struct {
  char *firstName;
  char *lastName;
  int nuid;
  double gpa;
} Student;
```

Code Sample 32: C Comparator Function Examples for `Student` structs

```c
/**
 * A comparator function to order Students by last name/first
    name
 * in alphabetic order
 */
int studentByNameCmp(const void *s1, const void *s2) {
```

```
6
7     const Student *a = (const Student *)s1;
8     const Student *b = (const Student *)s2;
9     int result = strcmp(a->lastName, b->lastName);
10    if(result == 0) {
11      return strcmp(a->firstName, b->firstName);
12    } else {
13      return result;
14    }
15  }
16
17  /**
18   * A comparator function to order Students by last name/first
        name
19   * in reverse alphabetic order
20   */
21  int studentByNameCmpDesc(const void *s1, const void *s2) {
22
23    int result = studentByNameCmp(s1, s2);
24    return -1 * result;
25  }
26
27  /*
28    Comparator function that orders by NUID in ascending
29    numerical order
30   */
31  int studentIdCmp(const void *s1, const void *s2) {
32
33    const Student *a = (const Student *)s1;
34    const Student *b = (const Student *)s2;
35    return (a->nuid - b->nuid);
36
37  }
```

E. Master Theorem

Theorem 1 (Master Theorem). *Let $T(n)$ be a monotonically increasing function that satisfies*

$$T(n) = aT\left(\frac{n}{b}\right) + f(n)$$

where $f(n) \in O(n^d)$ (that is, f is bounded by some polynomial) and $a \geq 1, b \geq 2, d > 0$.

$$T(n) = \begin{cases} O(n^d) & \text{if } a < b^d \\ O(n^d \log n) & \text{if } a = b^d \\ O(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

References

[1] Joshua Bloch. Extra, extra - read all about it: Nearly all binary searches and mergesorts are broken. http://googleresearch.blogspot.com/2006/06/extra-extra-read-all-about-it-nearly.html, 2006.

[2] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4(7):321–, July 1961.

[3] C. A. R. Hoare. Quicksort. *The Computer Journal*, 5(1):10–16, 1962.

[4] Donald E. Knuth. Von neumann's first computer program. *ACM Comput. Surv.*, 2(4):247–260, December 1970.

[5] Richard E. Pattis. Textbook errors in binary searching. In *Proceedings of the Nineteenth SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '88, pages 190–194, New York, NY, USA, 1988. ACM.

[6] Tim Peters. [Python-Dev] Sorting. Python-Dev mailing list, https://mail.python.org/pipermail/python-dev/2002-July/026837.html, July 2002.

[7] J. W. J. Williams. Algorithm 232: Heapsort. *Communications of the ACM*, 7(6):347–348, 1964.