CMPE 180-92

Data Structures and Algorithms in C++

November 9 Class Meeting

Department of Computer Engineering
San Jose State University

Fall 2017
Instructor: Ron Mak
www.cs.sjsu.edu/~mak

Assignment #11: Solution

```
vector<Word>& WordVector::get_data() { return data; }

int WordVector::get_size() const { return data.size(); }

int WordVector::get_count(const string text) const
{
    int index = find(text, 0, data.size()-1);
    return index >= 0 ? data[index].get_count() : -1;
}
```

Assignment #11: Solution, *cont'd*

```
Word *WordVector::insert(const string text)
{
    // First insertion?
    if (data.size() == 0)
    {
        data.push_back(Word(text));
        return &data[0];
    }
```

Assignment #11: Solution, *cont'd*

```
    // Insert a new word or increment the count of an existing word.
    else
    {
        // Look for the word in the vector.
        int index = find(text, 0, data.size()-1);

        // Not already in the vector: Scan the vector to insert the new word
        // at the appropriate position to keep the vector sorted.
        if (index < 0)
        {
            vector<Word>::iterator it = data.begin();
            while ((it != data.end()) && (text > it->get_text())) it++;
            it = data.insert(it, Word(text));
            return &(*it);
        }

        // Already in the vector: Increment the word's count.
        else
        {
            Word *word = &data[index];
            word->increment_count();
            return word;
        }
    }
}
```

Assignment #11: Solution, *cont'd*

```
Word *WordVector::search(const string text)
{
    // Look for the word in the vector.
    int index = find(text, 0, data.size()-1);

    // Return a pointer to the word if found, else nullptr.
    return index >= 0 ? &data[index] : nullptr;
}
```

Assignment #11: Solution, *cont'd*

```
int WordVector::find(const string text, int low, int high) const
{
    while (low <= high) {
        int mid = (low + high)/2;


        if (text == data[mid].get_text())
        {
            return mid;
        }
        else if (text < data[mid].get_text())
        {
            high = mid-1;
        }
        else
        {
            low = mid+1;
        }
    }

    return -1;
}
```

Assignment #11: Solution, *cont'd*

```
map<string, Word>& WordMap::get_data() { return data; }

int WordMap::get_size() const { return data.size(); }

int WordMap::get_count(const string text) const
{
    map<string, Word>::const_iterator it = data.find(text);
    return it != data.end() ? (it->second).get_count() : -1;
}
```

Assignment #11: Solution, *cont'd*

```
Word *WordMap::insert(const string text)
{
    // Look for the word in the map.
    map<string, Word>::iterator it = data.find(text);

    // Not already in the map: Enter the new word.
    if (it == data.end())
    {
        data[text] = Word(text);
        return &data[text];
    }

    // Already in the map: Increment the word's count.
    else
    {
        (it->second).increment_count();
        return &(it->second);
    }
}
```

Assignment #11: Solution, *cont'd*

```
Word *WordMap::search(const string text)
{
    // Look for the word in the map.
    map<string, Word>::iterator it = data.find(text);

    // Return a pointer to the word if found, else nullptr.
    return it != data.end() ? &(it->second) : nullptr;
}
```

Assignment #11: Solution, *cont'd*

```
class TimedContainer
{
public:
    TimedContainer();
    virtual ~TimedContainer();

    long get_elapsed_time() const;
    void reset_elapsed_time();

    Word *timed_insert(const string text);
    Word *timed_search(const string text);

    virtual Word *insert(const string text) = 0;
    virtual Word *search(const string text) = 0;

private:
    long elapsed_time;
};
```

Assignment #11: Solution, *cont'd*

```
Word *TimedContainer::timed_insert(const string text)
{
    // Start the timer.
    steady_clock::time_point start_time = steady_clock::now();

    // Do the insertion.
    Word *word = insert(text);

    // Compute the elapsed time in microseconds
    // and increment the total elapsed time.
    steady_clock::time_point end_time = steady_clock::now();
    long elapsed =
        duration_cast<microseconds>(end_time - start_time).count();
    elapsed_time += elapsed;

    return word;
}
```

Assignment #11: Solution, *cont'd*

```
Word *TimedContainer::timed_search(const string text)
{
    // Start the timer.
    steady_clock::time_point start_time = steady_clock::now();

    // Do the search.
    Word *word = search(text);

    // Compute the elapsed time in microseconds
    // and increment the total elapsed time.
    steady_clock::time_point end_time = steady_clock::now();
    long elapsed =
        duration_cast<microseconds>(end_time - start_time).count();
    elapsed_time += elapsed;

    return word;
}
```

A Nasty C++ Puzzle

Thing.h

```
#include <string>
#include <map>
using namespace std;

class Thing
{
public:
    Thing();
    virtual ~Thing();

    map<string, int> get_data();
    void insert(const string key, const int value);
    map<string, int>::iterator search(const string key);

private:
    map<string, int> data;
};
```

A Nasty C++ Puzzle, *cont'd*

Thing.cpp

```
#include <iostream>
#include "Thing.h"

Thing::Thing() {}
Thing::~Thing() {}

map<string, int> Thing::get_data() { return data; }

void Thing::insert(const string key, const int value)
{
    data[key] = value;
}

map<string, int>::iterator Thing::search(const string key)
{
    map<string, int>::iterator it = data.find(key);

    if (it != data.end()) cout << "(found)";
    else                  cout << "(not found)";

    return it;
}
```

San José State University

A Nasty C++ Puzzle, *cont'd*

IteratorEndTest.cpp

```
#include <iostream>
#include <string>
#include <map>
#include "Thing.h"

using namespace std;
int main()
{
    Thing t;

    t.insert("one", 1);
    t.insert("two", 2);
    map<string, int> tdata = t.get_data();
    map<string, int>::iterator it;

    cout << "Map dump:" << endl;

    for (it = tdata.begin(); it != tdata.end(); it++)
    {
        cout << it->first << ":" << it->second << endl;
    }
```

San José State University

A Nasty C++ Puzzle, *cont'd*

IteratorEndTest.cpp

```
    cout << endl << "Map searches:" << endl;

    cout << "  Searching for \"one\":";
    it = t.search("one");
    if (it != tdata.end()) cout << it->second << endl;
    else                   cout << "***" << endl;

    cout << "  Searching for \"two\":";
    it = t.search("two");
    if (it != tdata.end()) cout << it->second << endl;
    else                   cout << "***" << endl;

    cout << "  Searching for \"three\":";
    it = t.search("three");
    if (it != tdata.end()) cout << it->second << endl;
    else                   cout << "***" << endl;

    return 0;
}
```

A Nasty C++ Puzzle, *cont'd*

- Output:

```
Map dump:
one:1
two:2

Map searches:
  Searching for "one":(found)1
  Searching for "two":(found)2
  Searching for "three":(not found)1342860840
```

Why not *** ?

Sorting Algorithms

- There are several popular algorithms to sort a list of numbers.
  - selection sort
  - insertion sort
  - shellsort
  - quicksort
  - mergesort
- They differ in ease of programming and in efficiency.

Selection Sort

- The basic idea:
  - Make N-1 passes over a list of N numbers, starting with the position at index 0.
  - During each pass, find the smallest value from the unsorted values to put in the current position.

- Pseudocode:

After each pass, the size of the sorted part of the list grows by one.

```
for (pass = 0 through N-2)    // N-1 passes
{
   for (j = pass through N-1)    // unsorted part
   {
      find the smallest value among the list[j]
      elements and exchange it with list[pass]
   }
}
```

Selection Sort, *cont'd*

```
                4 6 9 3 0 1 1 6 7 6

After pass  1:  [ 0#] 6 9 3 4# 1 1 6 7 6

After pass  2:  [ 0 1#] 9 3 4 6# 1 6 7 6

After pass  3:  [ 0 1 1#] 3 4 6 9# 6 7 6

After pass  4:  [ 0 1 1 3#] 4 6 9 6 7 6

After pass  5:  [ 0 1 1 3 4#] 6 9 6 7 6

After pass  6:  [ 0 1 1 3 4 6#] 9 6 7 6

After pass  7:  [ 0 1 1 3 4 6 6#] 9# 7 6

After pass  8:  [ 0 1 1 3 4 6 6 6#] 7 9#

After pass  9:  [ 0 1 1 3 4 6 6 6 7#] 9
```

[...] is the sorted part.

marks the swapped elements

Insertion Sort

- One of the most simple and intuitive algorithms.
  - The way you would manually sort a deck of cards.
- Make N–1 passes over the list of data.
  - For pass $p = 1$ through N–1, the algorithm ensures that the data in positions 0 through p are sorted.

Original	34	8	64	51	32	21	Positions Moved
After $p = 1$	8	34	64	51	32	21	1
After $p = 2$	8	34	64	51	32	21	0
After $p = 3$	8	34	51	64	32	21	1
After $p = 4$	8	32	34	51	64	21	3
After $p = 5$	8	21	32	34	51	64	4

Figure 7.1 Insertion sort after each pass

Insertion Sort, *cont'd*

```
                    4  6*  9  3  0  1  1  6  7  6

After pass  1:   [ 4  6]  9*  3  0  1  1  6  7  6

After pass  2:   [ 4  6  9]  3*  0  1  1  6  7  6

After pass  3:   [ 3  4  6  9]  0*  1  1  6  7  6

After pass  4:   [ 0  3  4  6  9]  1*  1  6  7  6

After pass  5:   [ 0  1  3  4  6  9]  1*  6  7  6

After pass  6:   [ 0  1  1  3  4  6  9]  6*  7  6

After pass  7:   [ 0  1  1  3  4  6  6  9]  7*  6

After pass  8:   [ 0  1  1  3  4  6  6  7  9]  6*

After pass  9:   [ 0  1  1  3  4  6  6  6  7  9]
```

Insertion Sort, *cont'd*

- Insertion sort is inefficient because it swaps only adjacent values.
- A value may have to travel a long way through the array during a pass, one element at a time, to arrive at its proper place in the sorted part of the array.
- During the later passes, when the value being considered is toward the end of the array, that value potentially has to travel through more elements to arrive at its proper place.

Shellsort

- Like insertion sort, except we compare values that are h elements apart in the list.
  - h diminishes after completing a pass, for example, 5, 3, and 1.
- The final value of h must be 1, so the final pass is a regular insertion sort.
- The earlier passes get the array "nearly sorted" quickly.

Shellsort, *cont'd*

- After each pass, the array is said to be h_k-sorted.
  - Examples: 5-sorted, 3-sorted, etc.

Original	81	94	11	96	12	35	17	95	28	58	41	75	15
After 5-sort	35	17	11	28	12	41	75	15	96	58	81	94	95
After 3-sort	28	12	11	35	15	41	58	17	94	75	81	96	95
After 1-sort	11	12	15	17	28	35	41	58	75	81	94	95	96

Figure 7.3 Shellsort after each pass, using {1, 3, 5} as the increment sequence

Demo

Insertion Sort vs. Shellsort

- Shellsort is able to move a value a longer distance (h) without making the value travel through the intervening values.
- Early passes with large h make it easier for later passes with smaller h to sort.
- The final value of $h = 1$ is a simple insertion sort.
- Choosing a good increment sequence for h can produce a 25% speedup of the sort.

Suboptimal Shellsort

- The initial value of the diminishing interval h is half the size of the array.
- Each subsequent pass halves the interval.
- When h = 1, it's a regular insertion sort.

Suboptimal Shellsort, *cont'd*

```
      23 14 6 7 6 9 3 24 10 11 1 2 18 12 19 18 4 11 15 19 23 12 24 21 13

h = 12
      18# 14 6 7 6 9 3 24 10 11 1 2 23# 12 19 18 4 11 15 19 23 12 24 21 13
      18 12# 6 7 6 9 3 24 10 11 1 2 23 14# 19 18 4 11 15 19 23 12 24 21 13
      18 12 6 7 4# 9 3 24 10 11 1 2 23 14 19 18 6# 11 15 19 23 12 24 21 13
      18 12 6 7 4 9 3 19# 10 11 1 2 23 14 19 18 6 11 15 24# 23 12 24 21 13
      13# 12 6 7 4 9 3 19 10 11 1 2 18 14 19 18 6 11 15 24 23 12 24 21 23#

h = 6
      3# 12 6 7 4 9 13# 19 10 11 1 2 18 14 19 18 6 11 15 24 23 12 24 21 23
      3 12 6 7 1# 9 13 19 10 11 4# 2 18 14 19 18 6 11 15 24 23 12 24 21 23
      3 12 6 7 1 2# 13 19 10 11 4 9# 18 14 19 18 6 11 15 24 23 12 24 21 23
      3 12 6 7 1 2 13 14# 10 11 4 9 18 19# 19 18 6 11 15 24 23 12 24 21 23
      3 12 6 7 1 2 13 14 10 11 4 9 15# 19 19 18 6 11 18# 24 23 12 24 21 23
      3 12 6 7 1 2 13 14 10 11 4 9 15 19 19 12# 6 11 18 24 23 18# 24 21 23
```

Suboptimal Shellsort, *cont'd*

```
h = 3
     3 1# 6 7 12# 2 13 14 10 11 4 9 15 19 19 12 6 11 18 24 23 18 24 21 23
     3 1 2# 7 12 6# 13 14 10 11 4 9 15 19 19 12 6 11 18 24 23 18 24 21 23
     3 1 2 7 12 6 11# 14 10 13# 4 9 15 19 19 12 6 11 18 24 23 18 24 21 23
     3 1 2 7 4# 6 11 12 10 13 14# 9 15 19 19 12 6 11 18 24 23 18 24 21 23
     3 1 2 7 4 6 11 12 9# 13 14 10# 15 19 19 12 6 11 18 24 23 18 24 21 23
     3 1 2 7 4 6 11 12 9 12# 14 10 13 19 19 15# 6 11 18 24 23 18 24 21 23
     3 1 2 7 4 6 11 6# 9 12 12 10 13 14 19 15 19# 11 18 24 23 18 24 21 23
     3 1 2 7 4 6 11 6 9 12 12 10 13 14 11# 15 19 19# 18 24 23 18 24 21 23
     3 1 2 7 4 6 11 6 9 12 12 10 13 14 11 15 19 19 18 24 21# 18 24 23# 23

h = 1
     1# 3# 2 7 4 6 11 6 9 12 12 10 13 14 11 15 19 19 18 24 21 18 24 23 23
     1 2# 3# 7 4 6 11 6 9 12 12 10 13 14 11 15 19 19 18 24 21 18 24 23 23
     1 2 3 4# 7# 6 11 6 9 12 12 10 13 14 11 15 19 19 18 24 21 18 24 23 23
     1 2 3 4 6# 7# 11 6 9 12 12 10 13 14 11 15 19 19 18 24 21 18 24 23 23
     1 2 3 4 6 6# 7 11# 9 12 12 10 13 14 11 15 19 19 18 24 21 18 24 23 23
     1 2 3 4 6 6 7 9# 11# 12 12 10 13 14 11 15 19 19 18 24 21 18 24 23 23
     1 2 3 4 6 6 7 9 10# 11 12 12# 13 14 11 15 19 19 18 24 21 18 24 23 23
     1 2 3 4 6 6 7 9 10 11 11# 12 12 13 14# 15 19 19 18 24 21 18 24 23 23
     1 2 3 4 6 6 7 9 10 11 11 12 12 13 14 15 18# 19 19# 24 21 18 24 23 23
     1 2 3 4 6 6 7 9 10 11 11 12 12 13 14 15 18 19 19 21# 24# 18 24 23 23
     1 2 3 4 6 6 7 9 10 11 11 12 12 13 14 15 18 18# 19 19 21 24# 24 23 23
     1 2 3 4 6 6 7 9 10 11 11 12 12 13 14 15 18 18 19 19 21 23# 24 24# 23
     1 2 3 4 6 6 7 9 10 11 11 12 12 13 14 15 18 18 19 19 21 23 23# 24 24#
```

A More Optimal Shellsort

- Donald Knuth suggests the sequence of h values 1, 4, 13, 40, 121, ..., 3(i - 1) + 1.
  - Use the sequence in reverse.

A More Optimal Shellsort, *cont'd*

```
     23 14 6 7 6 9 3 24 10 11 1 2 18 12 19 18 4 11 15 19 23 12 24 21 13

h = 13
     12# 14 6 7 6 9 3 24 10 11 1 2 18 23# 19 18 4 11 15 19 23 12 24 21 13
     12 14 6 4# 6 9 3 24 10 11 1 2 18 23 19 18 7# 11 15 19 23 12 24 21 13
     12 14 6 4 6 9 3 23# 10 11 1 2 18 23 19 18 7 11 15 19 24# 12 24 21 13

h = 4
     6# 14 6 4 12# 9 3 23 10 11 1 2 18 23 19 18 7 11 15 19 24 12 24 21 13
     6 9# 6 4 12 14# 3 23 10 11 1 2 18 23 19 18 7 11 15 19 24 12 24 21 13
     6 9 3# 4 12 14 6# 23 10 11 1 2 18 23 19 18 7 11 15 19 24 12 24 21 13
     6 9 3 4 10# 14 6 23 12# 11 1 2 18 23 19 18 7 11 15 19 24 12 24 21 13
     6 9 3 4 10 11# 6 23 12 14# 1 2 18 23 19 18 7 11 15 19 24 12 24 21 13
     6 9 1# 4 10 11 3 23 12 14 6# 2 18 23 19 18 7 11 15 19 24 12 24 21 13
     6 9 1 2# 10 11 3 4 12 14 6 23# 18 23 19 18 7 11 15 19 24 12 24 21 13
     6 9 1 2 10 11 3 4 12 14 6 18# 18 23 19 23# 7 11 15 19 24 12 24 21 13
     6 9 1 2 7# 11 3 4 10 14 6 18 12 23 19 23 18# 11 15 19 24 12 24 21 13
     6 9 1 2 7 11 3 4 10 11# 6 18 12 14 19 23 18 23# 15 19 24 12 24 21 13
     6 9 1 2 7 11 3 4 10 11 6 18 12 14 15# 23 18 23 19# 19 24 12 24 21 13
     6 9 1 2 7 11 3 4 10 11 6 18 12 14 15 19# 18 23 19 23# 24 12 24 21 13
     6 9 1 2 7 11 3 4 10 11 6 18 12 12# 15 19 18 14 19 23 24 23# 24 21 13
     6 9 1 2 7 11 3 4 10 11 6 18 12 12 15 19 18 14 19 21# 24 23 24 23# 13
     6 9 1 2 7 11 3 4 10 11 6 18 12 12 15 19 13# 14 19 21 18 23 24 23 24#
```

A More Optimal Shellsort, *cont'd*

```
h = 1
      1# 6 9# 2 7 11 3 4 10 11 6 18 12 12 15 19 13 14 19 21 18 23 24 23 24
      1 2# 6 9# 7 11 3 4 10 11 6 18 12 12 15 19 13 14 19 21 18 23 24 23 24
      1 2 6 7# 9# 11 3 4 10 11 6 18 12 12 15 19 13 14 19 21 18 23 24 23 24
      1 2 3# 6 7 9 11# 4 10 11 6 18 12 12 15 19 13 14 19 21 18 23 24 23 24
      1 2 3 4# 6 7 9 11# 10 11 6 18 12 12 15 19 13 14 19 21 18 23 24 23 24
      1 2 3 4 6 7 9 10# 11# 11 6 18 12 12 15 19 13 14 19 21 18 23 24 23 24
      1 2 3 4 6 6# 7 9 10 11 11# 18 12 12 15 19 13 14 19 21 18 23 24 23 24
      1 2 3 4 6 6 7 9 10 11 11 12# 18# 12 15 19 13 14 19 21 18 23 24 23 24
      1 2 3 4 6 6 7 9 10 11 11 12 12# 18# 15 19 13 14 19 21 18 23 24 23 24
      1 2 3 4 6 6 7 9 10 11 11 12 12 15# 18# 19 13 14 19 21 18 23 24 23 24
      1 2 3 4 6 6 7 9 10 11 11 12 12 13# 15 18 19# 14 19 21 18 23 24 23 24
      1 2 3 4 6 6 7 9 10 11 11 12 12 13 14# 15 18 19# 19 21 18 23 24 23 24
      1 2 3 4 6 6 7 9 10 11 11 12 12 13 14 15 18 18# 19 19 21# 23 24 23 24
      1 2 3 4 6 6 7 9 10 11 11 12 12 13 14 15 18 18 19 19 21 23 23# 24# 24
```

Mergesort

- Divide and conquer!
- Divide
  - Split the list of values into two halves.
  - Recursively sort each of the two halves.
- Conquer
  - Merge the two sorted sublists back into a single sorted list.
- Nearly the optimal number of comparisons.

Mergesort for Linked Lists

- Mergesort does not rely on random access to the values in the list.
- Therefore, it is well-suited for sorting linked lists.

Mergesort for Linked Lists, *cont'd*

- How do we split a linked list into two sublists?
  - Splitting it at the midpoint may not efficient.
- Idea: Iterate down the list and assign the nodes alternating between the two sublists.
- Merging two sorted sublists should be easy.

Analysis of Mergesort

- How long does it take mergesort to run?
  - Let $T(N)$ be the time to sort N values.
  - It takes a constant 1 if $N = 1$.
  - It takes $T(N/2)$ to sort each half.
  - N to do the merge.
- Therefore, we have a recurrence relation:

$$T(N) = \begin{cases} 1 & \text{if } N = 1 \\ 2T(N/2) + N & \text{if } N > 1 \end{cases}$$

Analysis of Mergesort

- Solve:

$$T(N) = \begin{cases} 1 & \text{if } N = 1 \\ 2T(N/2) + N & \text{if } N > 1 \end{cases}$$

Assume N is a power of 2.

Divide both sides by N:

$$\frac{T(N)}{N} = \frac{T(N/2)}{N/2} + 1$$

Telescope: Since the equation is valid for any N that's a power of 2, successively replace N by $N/2$:

$$\frac{T(N/2)}{N/2} = \frac{T(N/4)}{N/4} + 1$$

$$\frac{T(N/4)}{N/4} = \frac{T(N/8)}{N/8} + 1$$

$$\frac{T(N/8)}{N/8} = \frac{T(N/16)}{N/16} + 1$$

$$\vdots$$

$$\frac{T(2)}{2} = \frac{T(1)}{1} + 1$$

Add together, and many convenient cancellations will occur.

Analysis of Mergesort

$$\frac{T(N)}{N} = T(1) + \log N$$

since there are $\log N$ number of 1's (remember that it's $\log_2$)

Multiply through by N:

$$T(N) = NT(1) + N\log N$$
$$= N + N\log N \quad \text{since } T(1) = 1$$
$$= O(N\log N)$$

- And so mergesort runs in $O(N \log N)$ time.

Break

Partitioning a List of Values

- "Divide and conquer" sorting algorithms require that the list first be split into smaller sublists that can be sorted separately.
- Then the sorted sublists can be recombined into a single sorted list.
  - The sublists are usually sorted using recursion.
- Are there better ways to partition (split) a list of values other than down the middle?

Partitioning a List of Values, *cont'd*

- Pick an arbitrary "pivot value" in the list.
- Move all the values less than the pivot value into one sublist.
- Move all the values greater than the pivot value into the other sublist.
- Now the pivot value is in its "final resting place".
  - It's in the correct position for the sorted list.
- Recursively sort the two sublists.
  - The pivot value doesn't move.
- Challenge: Find a good pivot value.

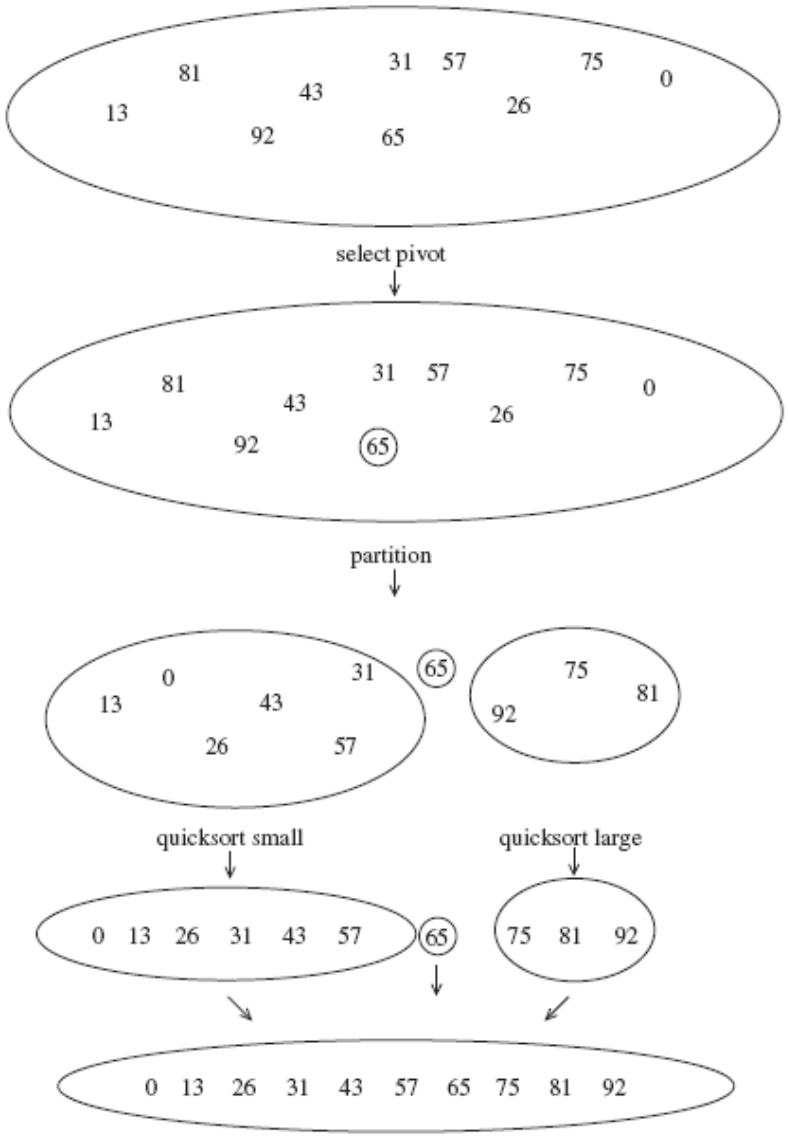

Figure 7.12 The steps of quicksort illustrated by example

Partition a List Using a Pivot

- Given a list, pick an element to be the pivot.
  - There are various strategies to pick the pivot.
  - The simplest is to pick the first element of the list.
- First get the chosen pivot value "out of the way" by swapping with the value currently at the right end.

```
6 1 4 9 0 3 5 2 7 8
```

```
8 1 4 9 0 3 5 2 7 6
```

Partition a List Using a Pivot, *cont'd*

```
8 1 4 9 0 3 5 2 7 6
```

- **Goal:** Move all values < pivot to the left part of the list and all values > pivot to the right part of the list.

Partition a List Using a Pivot, *cont'd*

- Set index `i` to the left end of the list and index `j` to one from the right end.

```
8 1 4 9 0 3 5 2 7 6
i               j
```

- While `i` < `j`:
  - Move `i` right, skipping over values < pivot.
    - Stop `i` when it reaches a value ≥ pivot.
  - Move `j` left, skipping over values > pivot.
    - Stop `j` when it reaches a value ≤ pivot.
  - After both `i` and `j` have stopped, swap the values at `i` and `j`.

Partition a List Using a Pivot, *cont'd*

Move `j`:

```
8 1 4 9 0 3 5 2 7 6
i             j
```

Swap:

```
2 1 4 9 0 3 5 8 7 6
i             j
```

Move `i` and `j`:

```
2 1 4 9 0 3 5 8 7 6
      i     j
```

Swap:

```
2 1 4 5 0 3 9 8 7 6
      i     j
```

Move `i` and `j`. They've crossed!

```
2 1 4 5 0 3 9 8 7 6
          j i
```

Swap the pivot with the `i`th element:

```
2 1 4 5 0 3 6 8 7 9
          j i
```

Now the list is properly partitioned for quicksort!

Quicksort

- A fast divide-and-conquer sorting algorithm.
  - A very tight and highly optimized inner loop.
  - Looks like magic in animation.
  - Average running time is $O(N \log N)$.
  - Worst-case running time is $O(N^2)$.
    - The worst case be made to occur very unlikely.

One of the most elegant and useful algorithms in computer science.

- Basic idea:
  - Partition the list using a pivot.
  - Recursively sort the two sublists.

Quicksort Pivot Strategy

- Quicksort is a fragile algorithm!
  - It is sensitive to picking a good pivot.
  - Attempts to improve the algorithm can break it.

- Simplest pivot strategy:
  Pick the first element of the list.
  - Worst strategy if the list is already sorted.
  - Running time $O(N^2)$.

Suboptimal Quicksort

- Choose the first element of the list as the pivot.

```
      5 24 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 1 8

SORTING from index 0 to 24
     [ 5 24 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 1 8]
Paritioning with pivot 5
     [ 8 24 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 1 5]
i = 0, j = 23, swapped 1 and 8
     [ 1# 24 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 8# 5]
i = 1, j = 19, swapped 2 and 24
     [ 1 2# 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24# 16 11 11 8 5]
i = 2, j = 1, swapped 8 and 2
     [ 1 2# 8# 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 5]
Partitioned with pivot 5 at index 2
     [ 1 2 5* 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 8]
```

Suboptimal Quicksort, *cont'd*

```
SORTING from index 0 to 1
     [ 1 2] 5 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 8
Paritioning with pivot 1
     [ 2 1] 5 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 8
i = 0, j = -1, swapped 2 and 24
     [ 2# 1] 5 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 8
Partitioned with pivot 1 at index 0
     [ 1* 2] 5 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 8
SORTED from index 0 to 1
     [ 1 2] 5 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 8

SORTING from index 3 to 24
     1 2 5 [ 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 8]
Paritioning with pivot 10
     1 2 5 [ 8 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 8 10]
i = 4, j = 23, swapped 8 and 23
     1 2 5 [ 8 8# 14 17 11 17 8 19 21 9 16 16 24 19 8 8 24 16 11 11 23# 10]
i = 5, j = 18, swapped 8 and 14
     1 2 5 [ 8 8 8# 17 11 17 8 19 21 9 16 16 24 19 8 14# 24 16 11 11 23 10]
i = 6, j = 17, swapped 8 and 17
     1 2 5 [ 8 8 8 8# 11 17 8 19 21 9 16 16 24 19 17# 14 24 16 11 11 23 10]
i = 7, j = 12, swapped 9 and 11
     1 2 5 [ 8 8 8 8 9# 17 8 19 21 11# 16 16 24 19 17 14 24 16 11 11 23 10]
i = 8, j = 9, swapped 8 and 17
     1 2 5 [ 8 8 8 8 9 8# 17# 19 21 11 16 16 24 19 17 14 24 16 11 11 23 10]
i = 9, j = 8, swapped 17 and 8
     1 2 5 [ 8 8 8 8 9 8# 17# 19 21 11 16 16 24 19 17 14 24 16 11 11 23 10]
Partitioned with pivot 10 at index 9
     1 2 5 [ 8 8 8 8 9 8 10* 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17]
```

San José State University

Suboptimal Quicksort, *cont'd*

```
SORTING from index 3 to 8
     1 2 5 [ 8 8 8 8 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Paritioning with pivot 8
     1 2 5 [ 8 8 8 8 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
i = 3, j = 6, swapped 8 and 8
     1 2 5 [ 8# 8 8 8# 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
i = 4, j = 5, swapped 8 and 8
     1 2 5 [ 8 8# 8# 8 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
i = 5, j = 4, swapped 8 and 8
     1 2 5 [ 8 8# 8# 8 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Partitioned with pivot 8 at index 5
     1 2 5 [ 8 8 8* 8 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17

SORTING from index 3 to 4
     1 2 5 [ 8 8] 8 8 9 8 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Paritioning with pivot 8
     1 2 5 [ 8 8] 8 8 9 8 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
i = 3, j = 3, swapped 8 and 8
     1 2 5 [ 8# 8] 8 8 9 8 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Partitioned with pivot 8 at index 3
     1 2 5 [ 8* 8] 8 8 9 8 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
SORTED from index 3 to 4
     1 2 5 [ 8 8] 8 8 9 8 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
```

Suboptimal Quicksort, *cont'd*

```
SORTING from index 6 to 8
     1 2 5 8 8 8 [ 8 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Paritioning with pivot 8
     1 2 5 8 8 8 [ 8 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
i = 6, j = 6, swapped 8 and 8
     1 2 5 8 8 8 [ 8# 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Partitioned with pivot 8 at index 6
     1 2 5 8 8 8 [ 8* 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17

SORTING from index 7 to 8
     1 2 5 8 8 8 8 [ 9 8] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Paritioning with pivot 9
     1 2 5 8 8 8 8 [ 8 9] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
i = 8, j = 7, swapped 9 and 8
     1 2 5 8 8 8 8 [ 8# 9#] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
Partitioned with pivot 9 at index 8
     1 2 5 8 8 8 8 [ 8 9*] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
SORTED from index 7 to 8
     1 2 5 8 8 8 8 [ 8 9] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
SORTED from index 6 to 8
     1 2 5 8 8 8 [ 8 8 9] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
SORTED from index 3 to 8
     1 2 5 [ 8 8 8 8 8 9] 10 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17
```

Suboptimal Quicksort, *cont'd*

```
SORTING from index 10 to 24
     1 2 5 8 8 8 8 8 9 10 [ 19 21 11 16 16 24 19 17 14 24 16 11 11 23 17]
Paritioning with pivot 19
     1 2 5 8 8 8 8 8 9 10 [ 17 21 11 16 16 24 19 17 14 24 16 11 11 23 19]
i = 11, j = 22, swapped 11 and 21
     1 2 5 8 8 8 8 8 9 10 [ 17 11# 11 16 16 24 19 17 14 24 16 11 21# 23 19]
i = 15, j = 21, swapped 11 and 24
     1 2 5 8 8 8 8 8 9 10 [ 17 11 11 16 16 11# 19 17 14 24 16 24# 21 23 19]
i = 16, j = 20, swapped 16 and 19
     1 2 5 8 8 8 8 8 9 10 [ 17 11 11 16 16 11 16# 17 14 24 19# 24 21 23 19]
i = 19, j = 18, swapped 24 and 14
     1 2 5 8 8 8 8 8 9 10 [ 17 11 11 16 16 11 16 17 14# 24# 19 24 21 23 19]
Partitioned with pivot 19 at index 19
     1 2 5 8 8 8 8 8 9 10 [ 17 11 11 16 16 11 16 17 14 19* 19 24 21 23 24]

SORTING from index 10 to 18
     1 2 5 8 8 8 8 8 9 10 [ 17 11 11 16 16 11 16 17 14] 19 19 24 21 23 24
Paritioning with pivot 17
     1 2 5 8 8 8 8 8 9 10 [ 14 11 11 16 16 11 16 17 17] 19 19 24 21 23 24
i = 17, j = 17, swapped 17 and 17
     1 2 5 8 8 8 8 8 9 10 [ 14 11 11 16 16 11 16 17# 17] 19 19 24 21 23 24
Partitioned with pivot 17 at index 17
     1 2 5 8 8 8 8 8 9 10 [ 14 11 11 16 16 11 16 17* 17] 19 19 24 21 23 24
```

Suboptimal Quicksort, *cont'd*

```
SORTING from index 10 to 16
     1 2 5 8 8 8 8 8 9 10 [ 14 11 11 16 16 11 16] 17 17 19 19 24 21 23 24
Paritioning with pivot 14
     1 2 5 8 8 8 8 8 9 10 [ 16 11 11 16 16 11 14] 17 17 19 19 24 21 23 24
i = 10, j = 15, swapped 11 and 16
     1 2 5 8 8 8 8 8 9 10 [ 11# 11 11 16 16 16# 14] 17 17 19 19 24 21 23 24
i = 13, j = 12, swapped 16 and 11
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11# 16# 16 16 14] 17 17 19 19 24 21 23 24
Partitioned with pivot 14 at index 13
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11 14* 16 16 16] 17 17 19 19 24 21 23 24

SORTING from index 10 to 12
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11] 14 16 16 16 17 17 19 19 24 21 23 24
Paritioning with pivot 11
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11] 14 16 16 16 17 17 19 19 24 21 23 24
i = 10, j = 11, swapped 11 and 11
     1 2 5 8 8 8 8 8 9 10 [ 11# 11# 11] 14 16 16 16 17 17 19 19 24 21 23 24
i = 11, j = 10, swapped 11 and 11
     1 2 5 8 8 8 8 8 9 10 [ 11# 11# 11] 14 16 16 16 17 17 19 19 24 21 23 24
Partitioned with pivot 11 at index 11
     1 2 5 8 8 8 8 8 9 10 [ 11 11* 11] 14 16 16 16 17 17 19 19 24 21 23 24
SORTED from index 10 to 12
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11] 14 16 16 16 17 17 19 19 24 21 23 24
```

Suboptimal Quicksort, *cont'd*

```
SORTING from index 14 to 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 [ 16 16 16] 17 17 19 19 24 21 23 24
Paritioning with pivot 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 [ 16 16 16] 17 17 19 19 24 21 23 24
i = 14, j = 15, swapped 16 and 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 [ 16# 16# 16] 17 17 19 19 24 21 23 24
i = 15, j = 14, swapped 16 and 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 [ 16# 16# 16] 17 17 19 19 24 21 23 24
Partitioned with pivot 16 at index 15
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 [ 16 16* 16] 17 17 19 19 24 21 23 24
SORTED from index 14 to 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 [ 16 16 16] 17 17 19 19 24 21 23 24
SORTED from index 10 to 16
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11 14 16 16 16] 17 17 19 19 24 21 23 24
SORTED from index 10 to 18
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11 14 16 16 16 17 17] 19 19 24 21 23 24

SORTING from index 20 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 19 24 21 23 24]
Paritioning with pivot 19
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 24 24 21 23 19]
i = 20, j = 19, swapped 24 and 19
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19# [ 24# 24 21 23 19]
Partitioned with pivot 19 at index 20
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 19* 24 21 23 24]
```

Suboptimal Quicksort, *cont'd*

```
SORTING from index 21 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 24 21 23 24]
Paritioning with pivot 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 24 21 23 24]
i = 21, j = 23, swapped 23 and 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 23# 21 24# 24]
i = 23, j = 22, swapped 24 and 21
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 23 21# 24# 24]
Partitioned with pivot 24 at index 23
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 23 21 24* 24]
```

Suboptimal Quicksort, *cont'd*

```
SORTING from index 21 to 22
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 23 21] 24 24
Paritioning with pivot 23
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 21 23] 24 24
i = 22, j = 21, swapped 23 and 21
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 21# 23#] 24 24
Partitioned with pivot 23 at index 22
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 21 23*] 24 24
SORTED from index 21 to 22
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 21 23] 24 24
SORTED from index 21 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 [ 21 23 24 24]
SORTED from index 20 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 19 21 23 24 24]
SORTED from index 10 to 24
     1 2 5 8 8 8 8 8 9 10 [ 11 11 11 14 16 16 16 17 17 19 19 21 23 24 24]
SORTED from index 3 to 24
     1 2 5 [ 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 23 24 24]
SORTED from index 0 to 24
     [ 1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 23 24 24]
```

Median-of-Three Pivot Strategy

- A good pivot value would be the median value of the list.
  - The median of a list of unsorted numbers is nontrivial to compute.
- Compromise:
  - Examine the two values at the ends of the list and the value at the middle position of the list.
  - Choose the value that's in between the other two.

Optimal Quicksort

- Median-of-three pivoting strategy.

```
 5 24 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 1 8

SORTING from index 0 to 24
     [ 5 24 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 1 8]
Paritioning with pivot 8
     [ 5 24 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 1 8]
i = 1, j = 23, swapped 1 and 24
     [ 5 1# 8 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 2 16 11 11 24# 8]
i = 2, j = 19, swapped 2 and 8
     [ 5 1 2# 10 23 14 17 11 17 8 19 21 9 16 16 24 19 8 8 8# 16 11 11 24 8]
i = 3, j = 18, swapped 8 and 10
     [ 5 1 2 8# 23 14 17 11 17 8 19 21 9 16 16 24 19 8 10# 8 16 11 11 24 8]
i = 4, j = 17, swapped 8 and 23
     [ 5 1 2 8 8# 14 17 11 17 8 19 21 9 16 16 24 19 23# 10 8 16 11 11 24 8]
i = 5, j = 9, swapped 8 and 14
     [ 5 1 2 8 8 8# 17 11 17 14# 19 21 9 16 16 24 19 23 10 8 16 11 11 24 8]
i = 6, j = 5, swapped 17 and 8
     [ 5 1 2 8 8 8# 17# 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 8]
Partitioned with pivot 8 at index 6
     [ 5 1 2 8 8 8 8* 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17]
```

Optimal Quicksort, *cont'd*

```
SORTING from index 0 to 5
     [ 5 1 2 8 8 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
Paritioning with pivot 5
     [ 2 1 8 8 8 5] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
i = 2, j = 1, swapped 8 and 1
     [ 2 1# 8# 8 8 5] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
Partitioned with pivot 5 at index 2
     [ 2 1 5* 8 8 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17

SORTING from index 0 to 1
     [ 2 1] 5 8 8 8 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
Paritioning with pivot 1
     [ 2 1] 5 8 8 8 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
i = 0, j = -1, swapped 2 and 393240
     [ 2# 1] 5 8 8 8 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
Partitioned with pivot 1 at index 0
     [ 1* 2] 5 8 8 8 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
SORTED from index 0 to 1
     [ 1 2] 5 8 8 8 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
```

Optimal Quicksort, *cont'd*

```
SORTING from index 3 to 5
     1 2 5 [ 8 8 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
Paritioning with pivot 8
     1 2 5 [ 8 8 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
i = 3, j = 4, swapped 8 and 8
     1 2 5 [ 8# 8# 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
i = 4, j = 3, swapped 8 and 8
     1 2 5 [ 8# 8# 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
Partitioned with pivot 8 at index 4
     1 2 5 [ 8 8* 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
SORTED from index 3 to 5
     1 2 5 [ 8 8 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
SORTED from index 0 to 5
     [ 1 2 5 8 8 8] 8 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17
```

Optimal Quicksort, *cont'd*

```
SORTING from index 7 to 24
     1 2 5 8 8 8 8 [ 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17]
Paritioning with pivot 17
     1 2 5 8 8 8 8 [ 11 17 14 19 21 9 16 16 24 19 23 10 8 16 11 11 24 17]
i = 8, j = 22, swapped 11 and 17
     1 2 5 8 8 8 8 [ 11 11# 14 19 21 9 16 16 24 19 23 10 8 16 11 17# 24 17]
i = 10, j = 21, swapped 11 and 19
     1 2 5 8 8 8 8 [ 11 11 14 11# 21 9 16 16 24 19 23 10 8 16 19# 17 24 17]
i = 11, j = 20, swapped 16 and 21
     1 2 5 8 8 8 8 [ 11 11 14 11 16# 9 16 16 24 19 23 10 8 21# 19 17 24 17]
i = 15, j = 19, swapped 8 and 24
     1 2 5 8 8 8 8 [ 11 11 14 11 16 9 16 16 8# 19 23 10 24# 21 19 17 24 17]
i = 16, j = 18, swapped 10 and 19
     1 2 5 8 8 8 8 [ 11 11 14 11 16 9 16 16 8 10# 23 19# 24 21 19 17 24 17]
i = 17, j = 16, swapped 23 and 10
     1 2 5 8 8 8 8 [ 11 11 14 11 16 9 16 16 8 10# 23# 19 24 21 19 17 24 17]
Partitioned with pivot 17 at index 17
     1 2 5 8 8 8 8 [ 11 11 14 11 16 9 16 16 8 10 17* 19 24 21 19 17 24 23]
```

Optimal Quicksort, *cont'd*

```
SORTING from index 7 to 16
     1 2 5 8 8 8 8 [ 11 11 14 11 16 9 16 16 8 10] 17 19 24 21 19 17 24 23
Paritioning with pivot 11
     1 2 5 8 8 8 8 [ 10 11 14 11 16 9 16 16 8 11] 17 19 24 21 19 17 24 23
i = 8, j = 15, swapped 8 and 11
     1 2 5 8 8 8 8 [ 10 8# 14 11 16 9 16 16 11# 11] 17 19 24 21 19 17 24 23
i = 9, j = 12, swapped 9 and 14
     1 2 5 8 8 8 8 [ 10 8 9# 11 16 14# 16 16 11 11] 17 19 24 21 19 17 24 23
i = 10, j = 10, swapped 11 and 11
     1 2 5 8 8 8 8 [ 10 8 9 11# 16 14 16 16 11 11] 17 19 24 21 19 17 24 23
Partitioned with pivot 11 at index 10
     1 2 5 8 8 8 8 [ 10 8 9 11* 16 14 16 16 11 11] 17 19 24 21 19 17 24 23

SORTING from index 7 to 9
     1 2 5 8 8 8 8 [ 10 8 9] 11 16 14 16 16 11 11 17 19 24 21 19 17 24 23
Paritioning with pivot 9
     1 2 5 8 8 8 8 [ 8 10 9] 11 16 14 16 16 11 11 17 19 24 21 19 17 24 23
i = 8, j = 7, swapped 10 and 8
     1 2 5 8 8 8 8 [ 8# 10# 9] 11 16 14 16 16 11 11 17 19 24 21 19 17 24 23
Partitioned with pivot 9 at index 8
     1 2 5 8 8 8 8 [ 8 9* 10] 11 16 14 16 16 11 11 17 19 24 21 19 17 24 23
SORTED from index 7 to 9
     1 2 5 8 8 8 8 [ 8 9 10] 11 16 14 16 16 11 11 17 19 24 21 19 17 24 23
```

Optimal Quicksort, *cont'd*

```
SORTING from index 11 to 16
     1 2 5 8 8 8 8 8 9 10 11 [ 16 14 16 16 11 11] 17 19 24 21 19 17 24 23
Paritioning with pivot 16
     1 2 5 8 8 8 8 8 9 10 11 [ 11 14 16 16 11 16] 17 19 24 21 19 17 24 23
i = 13, j = 15, swapped 11 and 16
     1 2 5 8 8 8 8 8 9 10 11 [ 11 14 11# 16 16# 16] 17 19 24 21 19 17 24 23
i = 14, j = 14, swapped 16 and 16
     1 2 5 8 8 8 8 8 9 10 11 [ 11 14 11 16# 16 16] 17 19 24 21 19 17 24 23
Partitioned with pivot 16 at index 14
     1 2 5 8 8 8 8 8 9 10 11 [ 11 14 11 16* 16 16] 17 19 24 21 19 17 24 23

SORTING from index 11 to 13
     1 2 5 8 8 8 8 8 9 10 11 [ 11 14 11] 16 16 16 17 19 24 21 19 17 24 23
Paritioning with pivot 11
     1 2 5 8 8 8 8 8 9 10 11 [ 11 14 11] 16 16 16 17 19 24 21 19 17 24 23
i = 11, j = 11, swapped 11 and 11
     1 2 5 8 8 8 8 8 9 10 11 [ 11# 14 11] 16 16 16 17 19 24 21 19 17 24 23
Partitioned with pivot 11 at index 11
     1 2 5 8 8 8 8 8 9 10 11 [ 11* 14 11] 16 16 16 17 19 24 21 19 17 24 23
```

Optimal Quicksort, *cont'd*

```
SORTING from index 12 to 13
     1 2 5 8 8 8 8 8 9 10 11 11 [ 14 11] 16 16 16 17 19 24 21 19 17 24 23
Paritioning with pivot 11
     1 2 5 8 8 8 8 8 9 10 11 11 [ 14 11] 16 16 16 17 19 24 21 19 17 24 23
i = 12, j = 11, swapped 14 and 11
     1 2 5 8 8 8 8 8 9 10 11 11# [ 14# 11] 16 16 16 17 19 24 21 19 17 24 23
Partitioned with pivot 11 at index 12
     1 2 5 8 8 8 8 8 9 10 11 11 [ 11* 14] 16 16 16 17 19 24 21 19 17 24 23
SORTED from index 12 to 13
     1 2 5 8 8 8 8 8 9 10 11 11 [ 11 14] 16 16 16 17 19 24 21 19 17 24 23
SORTED from index 11 to 13
     1 2 5 8 8 8 8 8 9 10 11 [ 11 11 14] 16 16 16 17 19 24 21 19 17 24 23

SORTING from index 15 to 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 [ 16 16] 17 19 24 21 19 17 24 23
Paritioning with pivot 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 [ 16 16] 17 19 24 21 19 17 24 23
i = 15, j = 15, swapped 16 and 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 [ 16# 16] 17 19 24 21 19 17 24 23
Partitioned with pivot 16 at index 15
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 [ 16* 16] 17 19 24 21 19 17 24 23
SORTED from index 15 to 16
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 [ 16 16] 17 19 24 21 19 17 24 23
SORTED from index 11 to 16
     1 2 5 8 8 8 8 8 9 10 11 [ 11 11 14 16 16 16] 17 19 24 21 19 17 24 23
SORTED from index 7 to 16
     1 2 5 8 8 8 8 [ 8 9 10 11 11 11 14 16 16 16] 17 19 24 21 19 17 24 23
```

Optimal Quicksort, *cont'd*

```
SORTING from index 18 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 [ 19 24 21 19 17 24 23]
Paritioning with pivot 19
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 [ 19 24 21 23 17 24 19]
i = 18, j = 22, swapped 17 and 19
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 [ 17# 24 21 23 19# 24 19]
i = 19, j = 18, swapped 24 and 17
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 [ 17# 24# 21 23 19 24 19]
Partitioned with pivot 19 at index 19
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 [ 17 19* 21 23 19 24 24]

SORTING from index 20 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 21 23 19 24 24]
Paritioning with pivot 21
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 19 23 24 24 21]
i = 21, j = 20, swapped 23 and 19
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 19# 23# 24 24 21]
Partitioned with pivot 21 at index 21
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 19 21* 24 24 23]
```

Optimal Quicksort, *cont'd*

```
SORTING from index 22 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 [ 24 24 23]
Paritioning with pivot 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 [ 23 24 24]
i = 23, j = 23, swapped 24 and 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 [ 23 24# 24]
Partitioned with pivot 24 at index 23
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 [ 23 24* 24]
SORTED from index 22 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 [ 23 24 24]
SORTED from index 20 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 [ 19 21 23 24 24]
SORTED from index 18 to 24
     1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 [ 17 19 19 21 23 24 24]
SORTED from index 7 to 24
     1 2 5 8 8 8 8 [ 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 23 24 24]
SORTED from index 0 to 24
     [ 1 2 5 8 8 8 8 8 9 10 11 11 11 14 16 16 16 17 17 19 19 21 23 24 24]
```

Quicksort, *cont'd*

- Quicksort doesn't do well for very short lists.
- When a sublist becomes too small, use another algorithm to sort the sublist such as insertion sort.

Assignment #12: Sorting Algorithms

- Implement sorting algorithms
  - selection
  - insertion
  - Shell (optimal and suboptimal)
  - quicksort (optimal and suboptimal)
  - mergesort
- Count the number of calls
  - constructors
  - copy constructors
  - destructors
- Compute elapsed times.

Sorting Animations

- https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
- http://www.sorting-algorithms.com