CSCI 104
Sorting Algorithms

Mark Redekopp
David Kempe
Algorithm Efficiency

SORTING
Sorting

• If we have an unordered list, sequential search becomes our only choice.
• If we will perform a lot of searches it may be beneficial to sort the list, then use binary search.
• Many sorting algorithms of differing complexity (i.e. faster or slower).
• Sorting provides a "classical" study of algorithm analysis because there are many implementations with different pros and cons.
Applications of Sorting

• Find the set_intersection of the 2 lists to the right
  – How long does it take?

• Try again now that the lists are sorted
  – How long does it take?
Sorting Stability

- A sort is stable if the order of equal items in the original list is maintained in the sorted list
  - Good for searching with multiple criteria
  - Example: Spreadsheet search
    - List of students in alphabetical order first
    - Then sort based on test score
    - I'd want student's with the same test score to appear in alphabetical order still

- As we introduce you to certain sort algorithms consider if they are stable or not
Bubble Sorting

- Main Idea: Keep comparing neighbors, moving larger item up and smaller item down until largest item is at the top. Repeat on list of size n-1
- Have one loop to count each pass, (a.k.a. i) to identify which index we need to stop at
- Have an inner loop start at the lowest index and count up to the stopping location comparing neighboring elements and advancing the larger of the neighbors
Bubble Sort Algorithm

```cpp
void bsort(vector<int> mylist)
{
    int i;
    for(i=mylist.size()-1; i > 0; i--){
        for(j=0; j < i; j++){
            if(mylist[j] > mylist[j+1]) {
                swap(j, j+1)
            }  
        }  }
}
```

**Pass 1**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
<td>8</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

**Pass 2**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td>no swap</td>
<td></td>
</tr>
</tbody>
</table>

**Pass n-2**

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td></td>
</tr>
<tr>
<td>j</td>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td>swap</td>
<td></td>
</tr>
</tbody>
</table>

...
Bubble Sort

Value

List Index

Courtesy of wikipedia.org
Bubble Sort Analysis

• Best Case Complexity:
  – When already ________________ but still have to ________________
  – O(____)

• Worst Case Complexity:
  – When ______________________
  – O(____)

```cpp
void bsort(vector<int> mylist)
{
    int i;
    for(i=mylist.size()-1; i > 0; i--){
        for(j=0; j < i; j++){
            if(mylist[j] > mylist[j+1]) {
                swap(j, j+1)
            }
        }
    }
}
```
Bubble Sort Analysis

• Best Case Complexity:
  – When already sorted (no swaps) but still have to do all compares
  – O(n²)

• Worst Case Complexity:
  – When sorted in descending order
  – O(n²)

```cpp
void bsort(vector<int> mylist)
{
  int i;
  for(i=mylist.size()-1; i > 0; i--){
    for(j=0; j < i; j++){
      if(mylist[j] > mylist[j+1]) {
        swap(j, j+1)
      }
    }
  }
}
```
Loop Invariants

• Loop invariant is a statement about what is true either before an iteration begins or after one ends

• Consider bubble sort and look at the data after each iteration (pass)
  – What can we say about the patterns of data after the k-th iteration?

```cpp
void bsort(vector<int> mylist)
{
    int i;
    for(i=mylist.size()-1; i > 0; i--){
        for(j=0; j < i; j++){
            if(mylist[j] > mylist[j+1]) {
                swap(j, j+1)
            }
        }
    }
}
```
Loop Invariants

• What is true after the k-th iteration?
• All data at indices n-k and above _____________
  – ∀i, i ≥ n – k:
• All data at indices below n-k are ______________
  ______________
  – ∀i, i < n – k:

```c
void bsort(vector<int> mylist)
{
    int i;
    for(i=mylist.size()-1; i > 0; i--){
        for(j=0; j < i; j++){
            if(mylist[j] > mylist[j+1]) {
                swap(j, j+1)
            }
        }
    }
}
```
Loop Invariants

- What is true after the k-th iteration?
- All data at indices n-k and above are sorted
  - \( \forall i, i \geq n - k: a[i] < a[i + 1] \)
- All data at indices below n-k are less than the value at n-k
  - \( \forall i, i < n - k: a[i] < a[n - k] \)

```c
void bsort(vector<int> mylist) {
    int i;
    for (i = mylist.size() - 1; i > 0; i--) {
        for (j = 0; j < i; j++) {
            if (mylist[j] > mylist[j + 1]) {
                swap(j, j + 1)
            }
        }
    }
}
```
Selection Sort

• Selection sort does away with the many swaps and just records where the min or max value is and performs one swap at the end
• The list/array can again be thought of in two parts
  – Sorted
  – Unsorted
• The problem starts with the whole array unsorted and slowly the sorted portion grows
• We could find the max and put it at the end of the list or we could find the min and put it at the start of the list
  – Just for variation let's choose the min approach
Selection Sort Algorithm

```cpp
void ssort(vector<int> mylist)
{
    for(i=0; i < mylist.size()-1; i++) {
        int min = i;
        for(j=i+1; j < mylist.size(); j++) {
            if(mylist[j] < mylist[min]) {
                min = j
            }
        }
        swap(mylist[i], mylist[min])
    }
}
```

Pass 1

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>1</td>
</tr>
</tbody>
</table>

Pass 2

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
</tr>
<tr>
<td>5</td>
<td>7</td>
</tr>
</tbody>
</table>

Pass n-2

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
Selection Sort

Value

List Index

Courtesy of wikipedia.org
Selection Sort Analysis

• Best Case Complexity:
  – ______________________
  – $O(\_\_\_\_\_)$
• Worst Case Complexity:
  – ______________________
  – $O(\_\_\_\_\_)$

```cpp
void ssort(vector<int> mylist)
{
    for(i=0; i < mylist.size()-1; i++){
        int min = i;
        for(j=i+1; j < mylist.size; j++){
            if(mylist[j] < mylist[min]) {
                min = j
            }
        }
    swap(mylist[i], mylist[min])
}
```
Selection Sort Analysis

• Best Case Complexity:
  – Sorted already
  – $O(n^2)$

• Worst Case Complexity:
  – When sorted in descending order
  – $O(n^2)$

```cpp
void ssort(vector<int> mylist)
{
    for(i=0; i < mylist.size()-1; i++){
        int min = i;
        for(j=i+1; j < mylist.size(); j++){
            if(mylist[j] < mylist[min]) {
                min = j
            }
        }
        swap(mylist[i], mylist[min])
    }
}
```
Loop Invariant

• What is true after the k-th iteration?

• All data at indices less than k are

________________________

– ∀i, i < k:

• All data at indices k and above are

________________________

________________________

– ∀i, i ≥ k:

```cpp
void ssort(vector<int> mylist) {
    for(i=0; i < mylist.size()-1; i++){
        int min = i;
        for(j=i+1; j < mylist.size(); j++){
            if(mylist[j] < mylist[min]) {
                min = j
            }
        }
        swap(mylist[i], mylist[min])
    }
}
```
Loop Invariant

• What is true after the k-th iteration?

• All data at indices less than k are sorted
  \[ \forall i, i < k: a[i] < a[i + 1] \]

• All data at indices k and above are greater than the value at k
  \[ \forall i, i \geq k: a[k] < a[i] \]

```cpp
void ssort(vector<int> mylist)
{
    for(i=0; i < mylist.size()-1; i++){
        int min = i;
        for(j=i+1; j < mylist.size(); j++){
            if(mylist[j] < mylist[min]) {
                min = j
            }   }
        swap(mylist[i], mylist[min])
    }
}
```
Insertion Sort Algorithm

• Imagine we pick up one element of the array at a time and then just insert it into the right position
• Similar to how you sort a hand of cards in a card game
  – You pick up the first (it is by nature sorted)
  – You pick up the second and insert it at the right position, etc.
Insertion Sort Algorithm

```cpp
void isort(vector<int> mylist)
{
    for(int i=1; i < mylist.size(); i++){
        int val = mylist[i];
        hole = i
        while(hole > 0 && val < mylist[hole-1]){
            mylist[hole] = mylist[hole-1];
            hole--;
        }
        mylist[hole] = val;
    }
}
```

Pass 1

```
7 3 8 6 5 1
h
7 7 8 6 5 1
h
3 7 8 6 5 1
h
```

Pass 2

```
3 7 8 6 5 1
val=3
3 7 8 6 5 1
h
3 7 8 6 5 1
h
```

Pass 3

```
3 7 8 6 5 1
val=8
3 7 8 6 5 1
h
3 7 7 8 5 1
h
3 6 7 8 5 1
h
```

Pass 4

```
3 6 7 8 5 1
val=6
3 6 7 8 5 1
h
3 6 7 8 8 1
h
3 6 6 7 8 1
h
3 5 6 7 8 1
h
```
Insertion Sort Analysis

• Best Case Complexity:
  – Sorted already
  – _______________

• Worst Case Complexity:
  – When sorted in descending order
  – _______________

```cpp
void isort(vector<int> mylist)
{  for(int i=1; i < mylist.size()-1; i++){
    int val = mylist[i];
    hole = i
    while(hole > 0 && val < mylist[hole-1]){  
      mylist[hole] = mylist[hole-1];
      hole--;
    }
    mylist[hole] = val;
  }
}```
Insertion Sort Analysis

• Best Case Complexity:
  – Sorted already
  – \(O(n)\)

• Worst Case Complexity:
  – When sorted in descending order
  – \(O(n^2)\)

```cpp
void isort(vector<int> mylist) {
    for(int i=1; i < mylist.size()-1; i++) {
        int val = mylist[i];
        hole = i;
        while(hole > 0 && val < mylist[hole-1]) {
            mylist[hole] = mylist[hole-1];
            hole--;
        }
        mylist[hole] = val;
    }
}```
Loop Invariant

- What is true after the k-th iteration?
- All data at indices less than ______________ – ∀i,
- Can we make a claim about data at k+1 and beyond?

```cpp
void isort(vector<int> mylist)
{
    for(int i=1; i < mylist.size()-1; i++)
    {
        int val = mylist[i];
        hole = i;
        while(hole > 0 && val < mylist[hole-1]){
            mylist[hole] = mylist[hole-1];
            hole--;
        }
        mylist[hole] = val;
    }
}
```
Loop Invariant

• What is true after the k-th iteration?
• All data at indices less than k+1 are sorted
  – $\forall i, i < k + 1: a[i] < a[i + 1]$
• Can we make a claim about data at k+1 and beyond?
  – No, it's not guaranteed to be smaller or larger than what is in the sorted list

```cpp
void isort(vector<int> mylist)
{
    for(int i=1; i < mylist.size()-1; i++){
        int val = mylist[i];
        hole = i
        while(hole > 0 && val < mylist[hole-1]){
            mylist[hole] = mylist[hole-1];
            hole--;
        }
        mylist[hole] = val;
    }
}
```
MERGESORT
Exercise

• [http://bits.usc.edu/websheets/?folder=cpp/cs104&start=merge&auth=Google#](http://bits.usc.edu/websheets/?folder=cpp/cs104&start=merge&auth=Google#)
  – merge
Merge Two Sorted Lists

- Consider the problem of merging two sorted lists into a new combined sorted list
- Can be done in $O(n)$
- Can we merge in place or need an output array?
Recursive Sort (MergeSort)

- Break sorting problem into smaller sorting problems and merge the results at the end
- MergeSort(0..n)
  - If list is size 1, return
  - Else
    - MergeSort(0..n/2 - 1)
    - MergeSort(n/2 .. n)
    - Combine each sorted list of n/2 elements into a sorted n-element list
Recursive Sort (MergeSort)

- Run-time analysis
  - # of recursion levels = Log₂(n)
  - Total operations to merge each level = n operations total to merge two lists over all recursive calls at a particular level
- Mergesort = O(n * log₂(n))
  - Usually has high constant factors due to extra array needed for merge
MergeSort Run Time

• Let's prove this more formally:
• $T(1) = \Theta(1)$
• $T(n) =$
Let's prove this more formally:

- \( T(1) = \Theta(1) \)
- \( T(n) = 2\times T(n/2) + \Theta(n) \)

\[
\begin{align*}
    k=1 & \quad T(n) = 2\times T(n/2) + \Theta(n) & \quad T(n/2) = 2\times T(n/4) + \Theta(n/2) \\
    k=2 & \quad = 2\times 2\times T(n/4) + 2\times \Theta(n) \\
    k=3 & \quad = 8\times T(n/8) + 3\times \Theta(n) \\
    & \quad = 2^k\times T(n/2^k) + k\times \Theta(n) \\
\end{align*}
\]

Stop @ \( T(1) \)
[i.e. \( n = 2^k \)]

\[
\begin{align*}
    k=\log_2 n & \quad = 2^k\times T(n/2^k) + k\times \Theta(n) = 2^{\log_2 n}\times \Theta(1) + \log_2 \times \Theta(n) = n + \log_2 \times \Theta(n) \\
    & \quad = \Theta(n\times \log_2 n)
\end{align*}
\]
Merge Sort

Value

List Index

Courtesy of wikipedia.org
Recursive Sort (MergeSort)

```cpp
void mergesort(vector<int>& mylist)
{
    vector<int> other(mylist);  // copy of array
    // use other as the source array, mylist as the output array
    msort(other, myarray, 0, mylist.size() );
}

void msort(vector<int>& mylist, vector<int>& output, int start, int end)
{
    // base case
    if(start >= end) return;
    // recursive calls
    int mid = (start+end)/2;
    msort(mylist, output, start, mid);
    msort(mylist, output, mid, end);
    // merge
    merge(mylist, output, start, mid, mid, end);
}

void merge(vector<int>& mylist, vector<int>& output, int s1, int e1, int s2, int e2)
{
    ...
}
```
Divide & Conquer Strategy

• Mergesort is a good example of a strategy known as "divide and conquer"

• 3 Steps:
  – Divide
    • Split problem into smaller versions (usually partition the data somehow)
  – Recurse
    • Solve each of the smaller problems
  – Combine
    • Put solutions of smaller problems together to form larger solution

• Another example of Divide and Conquer?
  – Binary Search
QUICKSORT
Partition & QuickSort

- Partition algorithm (arbitrarily) picks one number as the 'pivot' and puts it into the 'correct' location

```cpp
int partition(vector<int> mylist, int start, int end, int p)
{
    int pivot = mylist[p];
    swap(mylist[p], mylist[end]); // move pivot out of the way for now

    int left = start; int right = end-1;
    while(left < right){
        while(mylist[left] <= pivot && left < right)
            left++;
        while(mylist[right] >= pivot && left < right)
            right--;
        if(left < right)
            swap(mylist[left], mylist[right]);
    }
    if(mylist[right] > mylist[end]) {  // put pivot in
        swap(mylist[right], mylist[end]);  // correct place
        return right;
    }
    else { return end; }
}
```

Partition(mylist,0,5,5)

```
3 6 8 1 5 7
  l p
  r
3 6 8 1 5 7
  l p
3 6 5 1 8 7
  l p
3 6 5 1 8 7
  l p
3 6 5 7 8
  l p
Note: end is inclusive in this example
```
QuickSort

- Use the partition algorithm as the basis of a sort algorithm
- Partition on some number and the recursively call on both sides

```cpp
// range is [start,end] where end is inclusive
void qsort(vector<int>& mylist, int start, int end)
{
    // base case - list has 1 or less items
    if(start >= end) return;

    // pick a random pivot location [start..end]
    int p = start + rand() % (end+1);
    // partition
    int loc = partition(mylist,start,end,p)
    // recurse on both sides
    qsort(mylist,start,loc-1);
    qsort(mylist,loc+1,end);
}
```
Quick Sort

Value

List Index

Courtesy of wikipedia.org
QuickSort Analysis

• Worst Case Complexity:
  – When pivot chosen ends up being
    ________________
  – Runtime:

• Best Case Complexity:
  – Pivot point chosen ends up being the
    ________________
  – Runtime:
QuickSort Analysis

• **Worst Case Complexity:**
  – When pivot chosen ends up being min or max item
  – Runtime:
    • $T(n) = \Theta(n) + T(n-1)$

• **Best Case Complexity:**
  – Pivot point chosen ends up being the median item
  – Runtime:
    • Similar to MergeSort
    • $T(n) = 2T(n/2) + \Theta(n)$
QuickSort Analysis

• Average Case Complexity: $O(n \cdot \log(n))$
  – _____________ choose a pivot

3 6 8 1 5 7
QuickSort Analysis

- **Worst Case Complexity:**
  - When pivot chosen ends up being max or min of each list
  - $O(n^2)$

- **Best Case Complexity:**
  - Pivot point chosen ends up being the middle item
  - $O(n*\log(n))$

- **Average Case Complexity:** $O(n*\log(n))$
  - Randomly choose a pivot

- **Pivot and quicksort can be slower on small lists than something like insertion sort**
  - Many quicksort algorithms use pivot and quicksort recursively until lists reach a certain size and then use insertion sort on the small pieces
Comparison Sorts

• Big O of comparison sorts
  – It is mathematically provable that comparison-based sorts can never perform better than $O(n \cdot \log(n))$

• So can we ever have a sorting algorithm that performs better than $O(n \cdot \log(n))$?

• Yes, but only if we can make some meaningful assumptions about the input
OTHER SORTS
Sorting in Linear Time

• Radix Sort
  – Sort numbers one digit at a time starting with the least significant digit to the most.

• Bucket Sort
  – Assume the input is generated by a random process that distributes elements uniformly over the interval [0, 1)

• Counting Sort
  – Assume the input consists of an array of size N with integers in a small range from 0 to k.
Applications of Sorting

• Find the set_intersection of the 2 lists to the right
  – How long does it take?

• Try again now that the lists are sorted
  – How long does it take?
Other Resources

• http://www.youtube.com/watch?v=vxENKlcs2Tw


  ➢ http://www.math.ucla.edu/~rcompton/musical_sorting_algorithms/music_al_sorting_algorithms.html

• http://sorting.at/

• Awesome musical accompaniment: https://www.youtube.com/watch?v=ejpFmtYM8Cw