CS 310 – Advanced Data Structures and Algorithms

Sorting

June 7, 2018
Sorting

- One of the most fundamental problems in CS
- Input: a series of elements with a well-defined order
- Output: the elements listed according to this order
Topics

- Insertion sort
- Bubblesort
- Mergesort
- Quicksort
- Selectionsort
- Heapsort
void bubblesort(int A[], int n) {
    int i, j, temp;
    for (i = 0; i < n-1; i++) {
        boolean swapped = false;
        for (j = n-1; j > i; j--)
                swapped = true;
            }
        if(swapped == false) break;
    }
}
void insertionsort(int A[], int n) {
    for (int i = 1; i < n; i++) { /* n passes of loop */
        int key = A[i];
        /* Insert A[i] into the sorted sequence A[1 .. i - 1] */
        int j = i - 1;
        while( j >= 0 && A[j] > key){
            j = j - 1;
        }
        A[j + 1] = key;
    }
}
Best case: $O(n)$, when the input is sorted already
Worst case: $O(n^2)$, when the input is reverse-sorted
Average case: $O(n^2)$
For simplicity of analysis, assume there are no duplicates
Mergesort

- Divide and conquer
- 3 steps
  1. If the number of elements to sort is 0 or 1, return
  2. Recursively sort the first and second halves separately
  3. Merge the two sorted halves into a sorted sequence
- Mergesort is an $O(n \log n)$ algorithm
void sort(int[] A) {
    // check for empty or null array
    if (A==null || A.length==0) return;
    mergesort(A, 0, A.length - 1);
}
void mergesort(int A[], int l, int h) {
    if(l < h){
        int m = l+(h-l)/2; //Same as (l+h)/2, but avoids overflow
        mergesort(A, l, m);
        mergesort(A, m + 1, h);
        merge(A, l, m, h);
    }
}
void merge(int A[], int low, int middle, int high) {
    // Copy both parts into the helper array
    int[] helper = new int[A.length];
    for (int i = low; i <= high; i++) {
        helper[i] = A[i];
    }
    int i = low; int j = middle + 1; int k = low;
    while (i <= middle && j <= high) {
        if (helper[i] <= helper[j]) {
            A[k] = helper[i]; i++;
        } else {A[k] = helper[j]; j++; }
        k++;
    }
    // Copy the rest of the left side array into the target array
    while (i <= middle) {
        numbers[k] = helper[i]; k++; i++;
    }
}
Merge Sort example

image source: http://www.geeksforgeeks.org/merge-sort/
For simplicity, assume \( n \) is a power of 2

\[
T(n) = 2 \cdot T(n/2) + O(n) \\
= 2 \cdot (2 \cdot T(n/4) + O(n/2)) + O(n) \\
= 4 \cdot T(n/4) + O(n) + O(n) \\
= 4 \cdot (2 \cdot T(n/8) + O(n/4)) + O(n) + O(n) \\
= 8 \cdot T(n/8) + O(n) + O(n) + O(n) \\
\vdots \\
= 2^{\log n} \cdot T(n/2^{\log n}) + O(n) + O(n) + \cdots + O(n) \\
= n \cdot O(1) + O(n) \cdot \log n \\
= n \log n
\]
Quick sort

- Divide and conquer
- 4 steps
  1. If the number of elements in $S$ is 0 or 1, then return
  2. From $S$, pick any element $v$, called the pivot
  3. Partition $S - \{v\}$ into two disjoint groups: $L = \{x \in S - \{v\} \mid x \leq v\}$ and $R = \{x \in S - \{v\} \mid x \geq v\}$
  4. Return the result of Quicksort(L), followed by $v$, followed by Quicksort(R)

Note that after each partition, the pivot is in its final position in the sorted sequence (sometimes not true, for example, when choosing the middle element as pivot)
void sort(int[] A) {
    // check for empty or null array
    if (A==null || A.length==0) return;
    quicksort(A, 0, A.length - 1);
}

void quicksort(int A[], int low, int high) {
    int i = low, j = high;
    // Get the pivot element from the middle of the list
    int pivot = A[low + (high-low)/2];
    // Divide into two lists
    while (i <= j) {
        while (A[i] < pivot) i++;
        while (A[j] > pivot) j--;
        if (i <= j) {exchange(A, i, j);i++;j--;}
    }
    if (low < j) quicksort(A, low, j);
    if (i < high) quicksort(A, i, high);
}
Quick Sort (Using the last element as pivot)

```java
void sort(int[] A) {
    // check for empty or null array
    if (A==null || A.length==0) return;
    quicksort(A, 0, A.length - 1);
}
void quicksort(int A[], int low, int high) {
    if(low < high){
        int q = partition(A, low, high);
        quicksort(A, low, q - 1);
        quicksort(A, q + 1, high);
    }
}
```
Quick Sort (Using the last element as pivot)

```c
int partition(int A[], int low, int high){
    int x = A[high]; // x is the pivot
    int i = low - 1; // i is the "left-right boundary"
    int j = low;
    while (j < high){
        if(A[j] <= x){
            i += 1;
            exchange(A, i, j);
        }
        j += 1;
    }
    exchange(A, i+1, high);
    return i + 1;
}
```
Quicksort Example

figure 8.10
The steps of quicksort
Quicksort Performance

\[ T(n) = O(n) + T(L) + T(R) \]

- \( O(1) \) to pick a pivot
- The first term refers to the cost of partition, which is linear in \( n \)
- The second and third terms are recursive calls with \( L \) and \( R \)
- Best case: \( O(n \log n) \) when \( |L| \approx |R| \approx n/2 \)
- Worst case: \( O(n^2) \) when \( |R| = n - 1 \) or \( |L| = n - 1 \)

\[ T(n) = O(n) + T(n - 1) \]
Average Case of Quicksort

The average cost of a recursive call is

\[ T(L) = T(R) = \frac{T(0) + T(1) + T(2) + \ldots + T(n-1)}{n} \]

Thus

\[ T(n) = 2 \left( \frac{T(0) + T(1) + T(2) + \ldots + T(n-1)}{n} \right) + n \]

\[ nT(n) = 2(T(0) + T(1) + T(2) + \ldots + T(n-1)) + n^2 \]

\[ (n-1)T(n-1) = 2(T(0) + T(1) + T(2) + \ldots + T(n-2)) + (n-1)^2 \]

Take the difference

\[ nT(n) - (n-1)T(n-1) = 2T(n-1) + 2n - 1 \quad \text{(-1 is dropped)} \]

\[ nT(n) = (n+1)T(n-1) + 2n \]

\[ \frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1} \]
Telescoping Sum

\[
\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2}{n+1}
\]

\[
\frac{T(n-1)}{n} = \frac{T(n-2)}{n-1} + \frac{2}{n}
\]

\[
\frac{T(n-2)}{n-1} = \frac{T(n-3)}{n-2} + \frac{2}{n-1}
\]

\[\vdots\]

\[
\frac{T(2)}{3} = \frac{T(1)}{2} + \frac{2}{3}
\]
Add up all equations

\[
\frac{T(n)}{n+1} = \frac{T(1)}{2} + 2 \left( \frac{1}{3} + \frac{1}{4} + \ldots + \frac{1}{n} + \frac{1}{n+1} \right)
\]

\[
= 2 \left( 1 + \frac{1}{2} + \frac{1}{3} + \ldots + \frac{1}{n+1} \right) - \frac{5}{2}
\]

\[
= O(\log n)
\]

Note: harmonic series, \( \sum_{i=1}^{n} \frac{1}{i} \approx \ln n \)

Thus

\[
T(n) = O(n \log n)
\]
Picking the Pivot

- Choices of pivot: first, last element
  - Pick the first element, or the larger of the first two, or the last, or the smaller of the last two
  - If input is sorted or reverse sorted, all these are poor choices
- Pick the middle element
- Pick randomly
- Median-of-three
  - Use the median of the first, the middle, and the last elements
  - This strategy does not guarantee $O(n \log n)$ worst case, but it works well in practice

```c
int medianOf3(int a, int b, int c) {
  //a==0, b==1, c==2
  return a < b ? (b < c ? 1 : (a < c ? 2 : 0)) : (a < c ? 0 : (b < c ? 2 : 1));
}
```
Keys Equal to the Pivot

- As we move from left to right, incrementing $i$, should we stop when we encounter a key equal to the pivot?
- As we move from right to left, decrementing $j$, should we stop when we encounter a key equal to the pivot?
- Consider the case when all keys in the array are equal to the pivot.
- If we do not stop and keep incrementing $i$, it will reach the end of the array, resulting in imbalanced partition, worst case $O(n^2)$
- If we stop and swap identical keys, doing $O(n)$ redundant work, $i$ and $j$ will meet in the middle of the array, resulting in balanced partition, $O(n \log n)$
Quick Selection

• Selection: Find the \( k \)-th smallest element in an array of \( n \) elements
• Special case: Find the median, the \( \lfloor n/2 \rfloor \)-th smallest element
• Algorithm of \texttt{quickselect}(S, k)
  1. If the number of elements in \( S \) is 1, presumably \( k \) is also 1, so return the only element in \( S \)
  2. Pick any element \( v \) in \( S \), the pivot
  3. Partition \( S \) into \( L = \{ x \in S \setminus \{ v \} \mid x \leq v \} \) and \( R = \{ x \in S \setminus \{ v \} \mid x \geq v \} \)
  4. If \( k \) is exactly 1 more than \( |L| \), return the pivot
  5. If \( k \) is less than or equal to \( |L| \), call \texttt{quickselect}(L, k)
  6. Call \texttt{quickselect}(R, k - |L| - 1)

• Worst case \( O(n^2) \)
• Average case \( O(n) \)
Selection sort improves on the bubble sort by making only one exchange for every iteration.

Best, worst case: $O(n^2)$

```java
for (int i = 0; i < A.length - 1; i++){
    int index = i;
    for (int j = i + 1; j < A.length; j++)
        if (A[j] < A[index])
            index = j;
    exchange(A, i, index);
}
```
Heap Sort

- Heap sort is a comparison based sorting technique based on Binary Heap. It is similar to selection sort.
- Full binary tree
  - A binary tree in which every node other than the leaves has two children
- Complete binary tree
  - All leaves are on at most two adjacent levels
  - With the possible exception of the lowest level, all the levels are completed filled. The leaves on the lowest level are filled without gaps from the left.
- Heap (Binary heap)
  - A complete binary tree
  - Max heap: The value at each node is greater than or equal to the value in any descendant of that node. Min heap: The value at each node is less than or equal to the value in any descendant of that node.
Max Heap

- Because of the structure of a heap, it is most efficient to store a heap as an array.
- \{100,19,36,17,3,25,1,2,7\}

From Wikipedia
Heap Operation: Heapify

- Help to maintain heap property
- Heapify takes as input the array $A$ and one of the nodes $i$. Heapify($A$, $i$) assumes (and it must be true):
  - The tree rooted at $l = \text{LEFT}(i) = 2i + 1$ is a heap
  - The tree rooted at $r = \text{RIGHT}(i) = 2i + 2$ is a heap
- The subtree rooted at $i$ may violate the heap property
- Heapify is a procedure to let the value $A[i]$ “float down” to its proper position
- Takes $O(\log n)$ time, actually $O(\text{height of } A[i])$
void heapify(int A[], int i, int n) {
    int left = 2*i + 1;
    int right = 2*i + 2;
    int largest = i;
    if(left < n && A[left] > A[i]) //n is the size of the heap
        largest = left;
    if(right < n && A[right] > A[largest])
        largest = right;
    if(largest != i) {
        exchange(A, i, largest);
        heapify(A, largest, n);
    }
}
Heapify Example

CLRS figure 6.2 (All indices should be minus 1, since the array index starts from 0)
We can build a heap in a bottom-up manner by running `heapify()`.

All elements in range \( n/2 \) to \( n \) are heaps.

Walk backwards from \( n/2 - 1 \) to 0, calling `heapify` on each node, the order of processing guarantees that the children of node \( i \) are heaps when \( i \) is processed.

Takes \( O(n) \) time.

```java
void build_heap(int[] A){
    for(int i = A.length/2 - 1; i >= 0; i--)
        heapify(A, i);
}
```
Now we know how to build a heap, we can use it to actually sort an array in place, without using additional memory.

```c
void heapSort(int A[], int n){
    // One by one extract an element from heap (root - largest element)
    for (int i=n-1; i>=0; i--)
    {
        // Move current root to end
        exchange(A, 0, i);
        // call heapify on the reduced heap
        heapify(A, 0, i); //i is the size of the reduced heap
    }
}
```
Heaps are often used to implement priority queues.

A priority queue is a data structure that maintains a set \( S \) of elements, each with an associated value. The priority queue supports the following operations:

- **Insert** \( (S, x) \): insert the element \( x \) into the set \( S \)
- **Maximum** \( (S) \): returns the element of \( S \) with the largest value
- **Extract-max** \( (S) \): removes the element of \( S \) with the largest value
- **Increase-value** \( (S, x, k) \): increases the value of element \( x \) to the new value \( k \), which is assumed to be at least as large as \( x \)
Among their other applications, we can use max-priority queues to schedule jobs on a shared computer.

The max-priority queue keeps track of the jobs to be performed.

When a job is finished or interrupted, the scheduler selects the highest-priority job from among those pending by calling Extract-max.

The scheduler can add a new job to the queue at any time by calling Inset.
**CLRS 6.5**

**HEAP-EXTRACT-MAX**($A$)

1. if $A\.heap\text{-}size < 1$
2. 
   *error* "heap underflow"
3. $max = A[1]$
5. $A\.heap\text{-}size = A\.heap\text{-}size − 1$
6. **MAX-HEAPIFY**($A$, 1)
7. **return** $max$

**HEAP-INCREASE-KEY**($A$, $i$, $key$)

1. if $key < A[i]$
2. 
   *error* "new key is smaller than current key"
3. $A[i] = key$
4. while $i > 1$ and $A[\text{PARENT}(i)] < A[i]$
5. exchange $A[i]$ with $A[\text{PARENT}(i)]$
6. $i = \text{PARENT}(i)$

**MAX-HEAP-INSERT**($A$, $key$)

1. $A\.heap\text{-}size = A\.heap\text{-}size + 1$
2. $A[A\.heap\text{-}size] = −\infty$
3. **HEAP-INCREASE-KEY**($A$, $A\.heap\text{-}size$, $key$)
Comparison Sorts

- These algorithms share an interesting property: the sorted order they determine is based only on comparisons between the input elements.
- We call such sorting algorithms comparison sorts.
- Any comparison sort must make $\Omega(n \log n)$ comparisons in the worst case to sort $n$ elements.
The number of leaves: \( n! \)

The depth of a binary tree with \( L \) leaves is \( \Omega(\log n!) \)

\( \log n! = \Theta(n \log n) \)
Bucket Sort

- There are sorting algorithms that can sometimes be used and yield better than $\Omega(n \log n)$
- Sort $n$ numbers in range $k$. We will put each $A[i]$ to its appropriate bucket.
- Bucket $B[j]$ will contain numbers $A[i]$, such that $A[i]$ in range $\left(\frac{(j-1)k}{n}, \frac{jk}{n}\right)$
- Best case $O(n + k)$: the input is drawn from a uniform distribution
- Worst case $O(n^2)$
Bucket Sort

CLRS 8.4
Counting sort assumes that each of the \( n \) input elements is an integer in the range 0 to \( k \), for some integer \( k \).

When \( k = O(n) \), the sort runs in \( \Theta(n) \) time.

For each input element \( x \), counting sort determines the number of elements less than \( x \).

For example, if 17 elements are less than \( x \), then \( x \) belongs in output position 18.

Time complexity \( O(n + k) \). In practice, we usually use counting sort when have \( k = O(n) \), in which case running time is \( O(n) \).

Need two arrays: \( B \) holds the sorted output, \( C \) provides temporary working storage.
Counting Sort

**COUNTING-SORT**(\(A, B, k\))

1. let \(C[0..k]\) be a new array
2. for \(i = 0\) to \(k\)
   3. \(C[i] = 0\)
4. for \(j = 1\) to \(A\.length\)
   5. \(C[A[j]] = C[A[j]] + 1\)
   6. \// \(C[i]\) now contains the number of elements equal to \(i\).
5. for \(i = 1\) to \(k\)
   6. \(C[i] = C[i] + C[i - 1]\)
   7. \// \(C[i]\) now contains the number of elements less than or equal to \(i\).
6. for \(j = A\.length\) downto 1
   7. \(B[C[A[j]]] = A[j]\)
   8. \(C[A[j]] = C[A[j]] - 1\)

CLRS 8.2
Radix Sort

- What if the elements are in range from 0 to $n^2$?
- Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit.
- Space complexity $O(n + k)$
- Time complexity $O(wn)$ where $w$ is the number of digits (or words).

$$
\begin{array}{cccc}
329 & 720 & 720 & 329 \\
457 & 355 & 329 & 355 \\
657 & 436 & 436 & 436 \\
839 & 457 & 839 & 457 \\
436 & 657 & 355 & 657 \\
720 & 329 & 457 & 720 \\
355 & 839 & 657 & 839 \\
\end{array}
$$

CLRS 8.3
<table>
<thead>
<tr>
<th>Sorting Algorithms</th>
<th>Average</th>
<th>Best</th>
<th>Worst</th>
<th>Memory</th>
</tr>
</thead>
<tbody>
<tr>
<td>bubble sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insertion sort</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>selection sort</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>merge sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>quick sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2)$</td>
<td>$O(1)$ (?)</td>
</tr>
<tr>
<td>heap sort</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>bucket sort</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
<td>$O(n^2)$</td>
<td>$O(n)$</td>
</tr>
<tr>
<td>counting sort</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
<td>$O(n + k)$</td>
</tr>
<tr>
<td>radix sort</td>
<td>$O(wn)$</td>
<td>$O(wn)$</td>
<td>$O(wn)$</td>
<td>$O(n + k)$</td>
</tr>
</tbody>
</table>