Outline

HW3 is online. Due Tue, 11/20, 1 pm

Last time:

- Design Patterns: Brute-Force and Greedy
- Counting Change Problem
- Pattern Matching Algorithms as Examples of 2 Design Patterns
- Brute Force
- Simplified Boyer-Moore

Today: Sorting

- Selection sort
- Heap sort
- Insertion sort
Sorting

Objective: rearrange items in list such that their keys are ordered according to some well-defined ordering rule (e.g., numerical or alphabetical order)

Sort only when needed: if you just need to know where every item is, storing them in a dictionary (hash table or BST) will speed up lookup and allow for dynamic modification
Basic Operations and Concepts

CMP: compares items $A$ and $B$

SWAP: swaps items $A$ and $B$

(CMPSWAP: compares and swaps if $B$ is smaller than $A$)

Sort invariant: an item is sorted if it is either
- the last item in the list, or
- is $\leq$ (or $\geq$) the next item in the list

Order:
- ascending
- descending
- monotonically ascending or descending

Sugih Jamin (jamin@eecs.umich.edu)
Characteristics of Sorting Algorithms

- **internal** vs. **external** sorting: whether the whole data set can be kept in main memory. If only a block of data can be brought into memory at a time, access between blocks becomes expensive.

- **in place** vs. with **auxiliary** data structures: don’t forget the stack space used in recursive calls.

- by **comparison** or **distribution based** (later)

- **stable** vs. **unstable** sorting: do multiple items containing the same key keep their relative ordering after the sort (usually there is a secondary key whose ordering you want to keep, e.g., last name, first name, stable sort is thus useful for sorting over multiple keys).
Characteristics of Sorting Algorithms (contd)

- **direct vs. indirect**: indirect ones modify pointers/indices rather than the items themselves; useful when it is expensive to move items.

- **adaptive vs. non-adaptive**: whether sequence of operations performed is a function of given input data. Example: `bubble1()` is non-adaptive, `bubble2()` that stops when array is sorted is adaptive. Both are $O(n^2)$ algorithms.

- **simple vs. complex**: complex ones usually run faster but unstable and harder to implement.

- **worst-case vs. expected-case** performance.

Sugih Jamin (jamin@eecs.umich.edu)
Sorting Algorithms

Comparison based:

- selection sort: straight selection sort, heapsort
- insertion sort: insertion sort, shell sort
- merge sort
- exchange sort: bubble sort, quick sort

Distribution based: counting sort, radix sort, bucket sort
Selection Sort

Elements added to the sorted sequence in order

Algorithm (see Fig. 15.5):

- given a “pile” of items in an array
- find the smallest (largest) item in array, swap with first (last) position
- find the second smallest (largest) item in array, swap with second (second last) position
- non-adaptive version: continue until end of array
- adaptive version: don’t swap if item is already in correct position

Straight selection sort: performs a linear search for the next smallest (largest) item

Running time:
Selection Sort

Sort array $a[5, 8, 2, 5, 3, 1]$

Non-adaptive:

```c
void ssort_na(int *a, int N)
```

Adaptive:

```c
void ssort_a(int *a, int N)
```

Is it stable?

Sugih Jamin (jamin@eecs.umich.edu)
Heapsort

Use a MaxHeap (MinHeap) to “select” the next largest (smallest) element

First build unsorted sequence into a binary MaxHeap (MinHeap)

MaxHeap property: root must be larger than all children

Since a heap is a complete binary tree,
- can be implemented as an array
- sorting can be done in place

Given an array of unsorted elements, first “heapify” the array
Heapify: Building a Heap

Bottom-up heapification, visit tree post-order:

- heapify left child
- heapify right child
- heapify root node

To heapify a tree when both children are already heapified: 
**percolate down** root node
(repeatedly swap root node with max (min) of two children until both children are smaller (larger) than root node or root node has become a leaf)
In-place Heapsort

Heapify (Fig. 15.7 Preiss):

- an $n$-node complete binary tree has $\lceil n/2 \rceil$ leaf nodes
- only need to percolate down elements at indices $\lfloor n/2 \rfloor$ to 1 (bottom up)

Heapsort (Fig. 15.8 Preiss): next extract the elements in order

- max item is always at root, index 1
- rightmost element is always at the last index
- after removing the root, move rightmost element to index 1 and percolate down
- store max element at the now free rightmost slot
Heapsort Time Complexity

Heapify phase: $O(n)$

- percolate down is $O(\log n)$
- inspecting $\lfloor n/2 \rfloor$ element is $O(n)$
- complexity appears to be $O(n \log n)$, but is actually $O(n)$, (Preiss Th. 15.2)

Heapsort phase: $O(n \log n)$

- must dequeue $n$ elements,
- each `dequeue_max()` takes $O(\log n)$

Total time complexity is still $O(n \log n)$
Insertion Sort

Algorithm (see Fig. 15.2):
- given a “pile” of items in an array
- insert into an already sorted array one item at a time using linear search
- non-adaptive version: continues linear search till end of sorted array
- adaptive version: stops linear search once an insertion point has been found

Contrast to selection sort:
- selection sort: select item from pile in order, add to the front (end) of sorted list
- insertion sort: pick next item from pile, insert in order into sorted list
Insertion Sort

Sort array $a[5, 2, 3, 8, 5, 1]$:

Non-adaptive:

```c
void
isort_na(int *a, int N)
```

Is it stable?

Adaptive:

```c
void
isort_a(int *a, int N)
```

Is it stable?
Running Time and Optimizations

Running time:

“Optimizations”:

- binary search instead of linear search?
- move instead of swap: saves $O(n^2)$ copies
- use of sentinel: saves $O(n^2)$ CMPs