bigO,c++,lecture10,sorting

Carey's 2 Rules of Sorting

1. Don't choose a sorting algorithm until you **understand the requirements** of your problem.
2. Always choose the **simplest** sorting algorithm that meets your requirements.

Sorting Overview

Selection Sort	Unstable	Always $O(n^2)$, but simple to implement. Can be used with linked lists. Minimizes the number of item-swaps (important if swaps are slow)
Insertion Sort	Stable	$O(n)$ for already or nearly-ordered arrays. $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement.
Bubble Sort	Stable	$O(n)$ for already or nearly-ordered arrays (with a good implementation). $O(n^2)$ otherwise. Can be used with linked lists. Easy to implement. Rarely a good answer on an interview!
Shell Sort	Unstable	$O(n^{1.25})$ approx. OK for linked lists. Used in some embedded systems (eg, in a car) instead of quicksort due to fixed RAM usage.
Quick Sort	Unstable	$O(n \log_2 n)$ average, $O(n^2)$ for already/mostly/reverse ordered arrays or arrays with the same value repeated many times. Can be used with linked lists. Can be parallelized across multiple cores. Can require up $O(n)$ slots of extra RAM (for recursion) in the worst case, $O(\log_2 n)$ avg.
Merge Sort	Stable	$O(n \log_2 n)$ always. Used for sorting large amounts of data on disk (aka "external sorting"). Can be used to sort linked lists. Can be parallelized across multiple cores. Downside: Requires n slots of extra memory/disk for merging – other sorts don't need xtra RAM.
		$O(n \log_2 n)$ always. Sometimes used in low-RAM embedded

Heap Sort	Unstable	systems because of its performance/low memory req’ts.

Sort Stability

- A sort is stable if it doesn’t switch up the order of matching items
- For example, if we sort the numbers below...
- 8 **8** 7 4 3
- The resulting order should have the bold 8 still be on the right
- 3 4 7 8 **8**
- in order to maintain a “stable” sort

Selection Sort

- can work with a linked list
- **unstable** sort (due to items getting swapped)

Algorithm

Look at all N items, select the smallest item
Swap this with the first item
Repeat this process with the remaining N-1 items (not including the first one, that is)

Effeciency

- $O(N^2)$ **always**
- (pretty slow)

When to Use

- too slow to be used in most real-world applications

```
void selectionSort(int A[], int n) {
  for (int i = 0; i < n; i++){
    int minIndex = i;
    for (int j = i+1; j < n; j++){
      if (A[j] < A[minIndex])
        minIndex = j;
    }
    swap(A[i], A[minIndex]);
  }
}
```

Insertion Sort

- can be used in linked lists that have “previous” pointers
    - needs to be doubly-linked
- **stable** sort

Algorithm

Start with a set size **S** of 2.

While there are still books to sort,
  Focus on the first **S** books.
  If the last book in this set is in the wrong order,
    Remove it from the shelf.
    Shift the books before it to the right, as necessary.
    Insert our book into the proper slot.
  Increment the set size **S**.

Effeciency

- O(N^2) at worst (if items are in full reverse order)
- O(N) at best (if items are already in order)

When to Use

- too slow to be used in most real-world applications

```
void insertionSort(int A[], int n) {
   for(int s = 2; s <= n; s++) {
      int sortMe = A[ s - 1 ];
      int i = s - 2;
      while (i >= 0 && sortMe < A[i]) {
         A[i+1] = A[i];
          --i;
      }
      A[i+1] = sortMe;
   }
}
```

Bubble Sort

- can be used in linked lists
- **stable** sort

Algorithm

Start at the top element of your array
Comapre the first two elements.
If they are out of order, swap them.
  Advance one element in your array.
Repeat the compare/swap/advance steps until you hit the end.
When you hit the end, if you have made at least one swap, repeat the whole process.

Effeciency

- O(N^2) at worst
- really effecient on pre-sorted or almost sorted arrays
    - as low as O(N)

When to Use

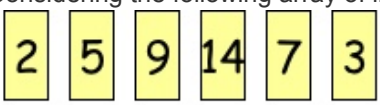
- too slow to be used in most real-world applications

```
void bubbleSort(int Arr[], int n) {
  bool atLeastOneSwap;
  do {
    atLeastOneSwap = false;
    for (int j = 0; j < (n-1); j++)
```

```
      if (Arr[j] > Arr[j + 1]) {
        Swap(Arr[j],Arr[j+1]);
        atLeastOneSwap = true;
      }
    }
    while (atLeastOneSwap == true);
}
```

Sorting Challenge

Considering the following array of integers



which has been sorted by one round of either **selection**, **insertion**, or **bubble** sort

Which of these sorts could **NOT** have been used on this array?

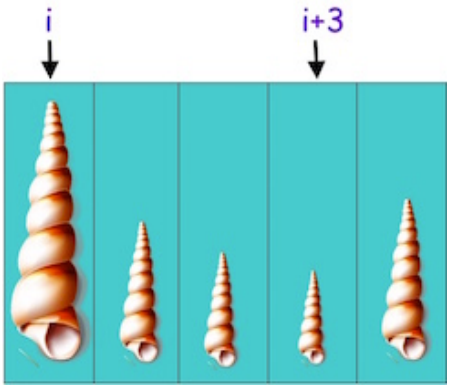
- **Solution**
  - **bubble sort**: the 14 must be at the end after one round
- **NOT Solution**
  - **selection sort**: one round of selection sort would bring the minimum item to the front (the 2)
  - **insertion sort**: a single round would cause the first two items to be sorted, which they are

H-Sort

- If a data structure is h-sorted, every element is less than the element h items away
- a 1-sort is the **same as a bubble sort**

Algorithm

Pick a value of h.
For each element in the array:
  If `A[i]` and `A[i+h]` are out of order
    Swap the two elements.
  If you swapped any elements during the last pass
    Repeat the entire process again



Shell Sort

- can be applied to a link list using two pointers
  - one pointing to the start, one pointing to h nodes ahead

- h-sort **repeatedly** with decreasing h-values, **ending with h = 1**
- each h-sort more correctly sorts the data
- **unstable** sort

Algorithm

1. Select a sequence of decreasing h values (e.g. 8, 4, 2, 1)
2. Completely 8-sort the array
3. Completely 4-sort the array
4. Completely 2-sort the array
5. Finally bubble sort (1-sort) the array
6. Done! Fully sorted!

Effeciency

- $O(N^2)$ at worst
- average case has not been determined mathematically
- experimental measurements suggest it is $O(N^{1.25})$

When to Use

- often used in systems with low memory since shell sort uses a fixed amount of memory

Quick Sort

- **Divide and conquer strategy**
- can be applied to a doubly-linked list
- **unstable** because the partition function may swap elements
- uses a variable amount of RAM depending on max recursion depth
    - pre-sorted array will use O(N) space due to N-deep recursion
    - BAD — especially for low-memory machines
    - most real implementations address this issue through some clever enhancements to the basic algorithm we use

Algorithm

1. If the array contains only 0 or 1 element, return.
2. Select an arbitrary element P from the array (typically the first element).
3. Move all elements that are less than or equal to P to the left of the array and all elements that are greater than P to the right (called **partitioning**).
4. Recursively repeat this process on the left sub-array and then the right sub-array.

Partition Function (takes N steps)

- uses the first item as the pivot value and moves smaller items to the left and everything else to the right
- returns the index of the pivot value

Effeciency

- $O(N \bullet \log_2 N)$
- slows down with **already sorted data**
    - 

    - Partitioning this array in half doesn't **divide** the problem at all!
    - big-O ends up being $O(N^2)$
- same problem with **reverse sorted data**

When to Use

- variants of quicksort are used everywhere
- can be parallelized across multiple cores

```
void QuickSort(int Array[],int First,int Last) {
  if (Last – First >= 1 ) {
    int PivotIndex;
    PivotIndex = Partition(Array,First,Last);
    QuickSort(Array,First,PivotIndex-1); // left
    QuickSort(Array,PivotIndex+1,Last);  // right
  }
}
```

Merge Sort

- **Divide and conquer strategy**
- extremely efficient sort
- involves taking **two arrays as input** and outputting a combined, **third sorted array**
- can be applied

Algorithm (recursive)

1. If array has one element, then return (it's sorted).
2. Split up the array into two equal sections.
3. Recursively call Mergesort function on the left half
4. Recursively call Mergesort function on the right half
5. Merge the two halves using our merge function

Partition Function (takes N steps)

- uses the first item as the pivot value and moves smaller items to the left and everything else to the right
- returns the index of the pivot value

Effeciency

- $O(N \bullet \log_2 N)$
- works equally well regardless of the ordering of the data
- however, merge function needs secondary arrays to merge, this can slow things down
    - in contrast, **quick sort** does not need to allocate any new arrays to work

When to Use

- used for sorting large amounts of data on disk (aka "external sorting")
- can be parallelized across multiple cores

Downsides

- requires N slots of extra memory