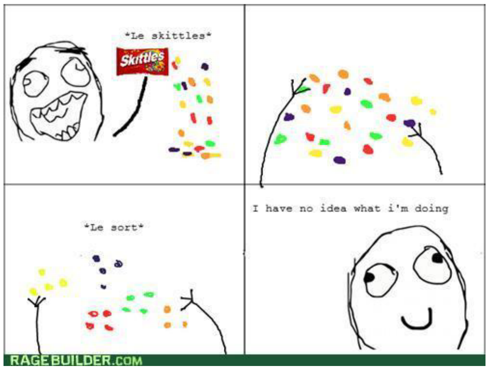
Adam Blank Lecture 12 Autumn 2015

CSE 332

Data Abstractions

CSE 332: Data Abstractions

Sorting



Why Study Sorting?

A Useful Invariant

- Binary Search only works **if the array is sorted**
- BSTs are **based around the idea of sorting the input**

"Local" vs. "Global" Views of Data

- All of our data structure so far only gave us a local view:
  - Heaps gave us a view of the max or min
  - Stacks and Queues gave us a view of most/least recent
  - Dictionaries give us a view of "associated data"
- A "global" view tells us how the elements all interact with each other
- There is no "best" sorting algorithm: most sorts have a purpose

What is SORT?

SORT is the computational problem with the following requirements:

Inputs

- An array `A` of `E` data of length L.
- A consistent, total ordering on all elements of type `E`:

`compare(a, b)`

Post-Conditions

- For all $0 \le i < j < L$, `A[i]` $\le$ `A[j]`
- Every element originally in the array must be somewhere in the resulting array.

An algorithm that solves this computational problem is called a **Comparison Sort**.

Properties of Sorting Algorithms

There are several important properties sorting algorithms

Definition (In-Place Sorting)

A sorting algorithm is **in-place** if we don't require (more than $\mathcal{O}(1)$) extra space to do the sort.
It's a useful property, because:

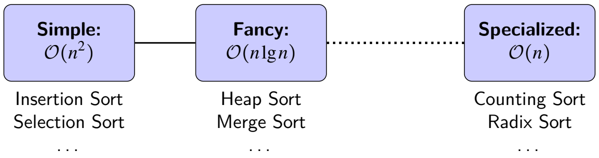
- The less memory we use the better...

Definition (Stable Sorting)

A sorting algorithm is **stable** if the order of any **equal** elements remains the same.
It's a useful property, because:

- We often want to first sort by one index and then another.
- Two objects might be equal but not completely duplicates.

Spectrum of Sorting



Simple: $\mathcal{O}(n^2)$
Insertion Sort
Selection Sort
...

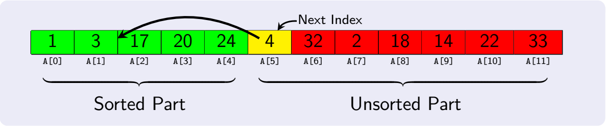
Fancy: $\mathcal{O}(n \lg n)$
Heap Sort
Merge Sort
...

Specialized: $\mathcal{O}(n)$
Counting Sort
Radix Sort
...

There are a lot of different sorting algorithms out there!

We're not going to cover **all** of them, but we will cover the ones that demonstrate clear advantages in one way or another.

Simple Sorting: Insertion Sort



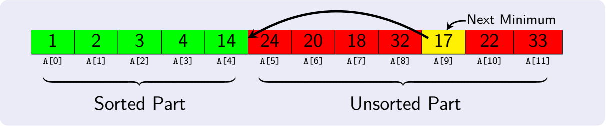
Algorithm

```
// i is "# of elements sorted"
for (i = 0; i < n; i++) {
   swap(i, findPlace(i));
}
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

Simple Sorting: Selection Sort



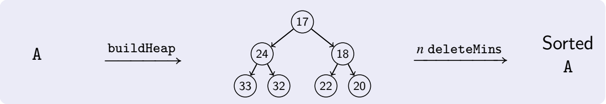
Algorithm

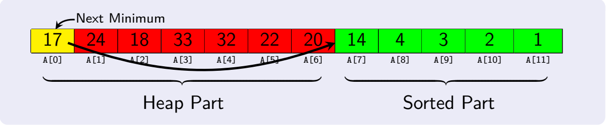
```
// i is "# of elements sorted"
for (i = 0; i < n; i++) {
   swap(i, findMin(i, n));
}
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

Fancy Sorting: Heap Sort





Algorithm

```
E[] A = buildHeap();
for (i = 0; i < n; i++) {
   swap(n − i − 1, A.deleteMin());
}
```

Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

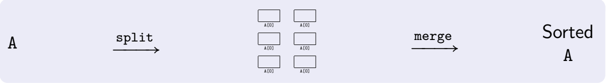
Divide and Conquer

Divide and Conquer is a very useful algorithmic technique. It consists of multiple steps:

1. **Divide** the input into smaller pieces (recursively)
2. **Conquer** the individual pieces as base cases
3. **Combine** the finished pieces together (recursively)

```
algorithm(input) {
   if (small enough) {
      return conquer(input);
   }
   pieces = divide(input);
   for (piece in pieces) {
      result = combine(result, algorithm(piece));
   }
   return result;
}
```

Fancy Sorting: Merge Sort



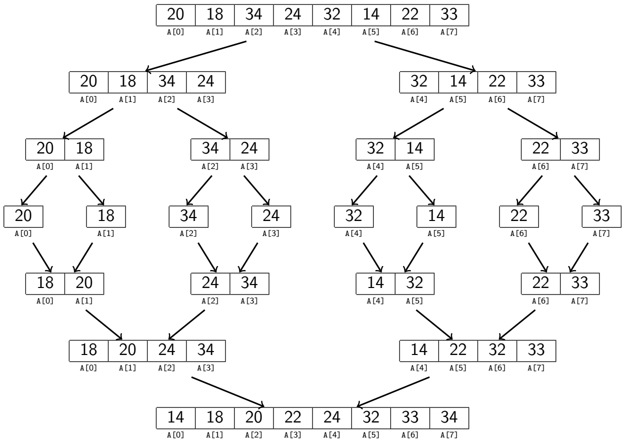
Algorithm

```
sort(A) {
   if (A.length < 2) {
      return A;
   }
   return merge(
      sort(A[0, ..., mid]),
      sort(A[mid + 1, ...])
   );
}
```

Runtime and Analysis

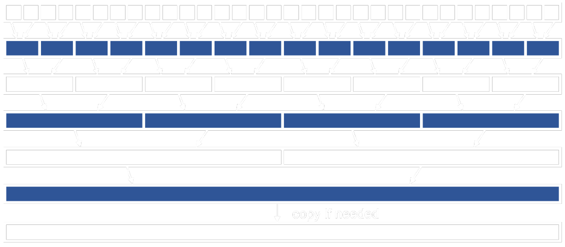
- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

Fancy Sorting: Merge Sort



Fancy Sorting: Merge Sort

The standard merge sort copies the array **at every step**. This is super slow! We can do better.



In this version, we allocate a **single auxiliary array** and swap between it and the original on each stage.

This is easier iteratively!

Fancy Sorting: Merge Sort (Linked Lists & Big Data)

In general, we've been sorting with **arrays**, but what about **linked lists?**

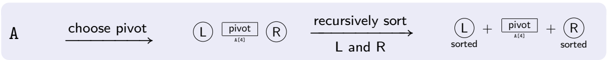
An Approach

- Convert to an array ($\mathcal{O}(n)$)
- Sort ($\mathcal{O}(n \lg(n))$)
- Convert to a list ($\mathcal{O}(n)$)

But, we can actually **do merge sort directly on a list**! (This is not true for heapsort or quicksort!)

Mergesort is also a good choice for external sorting, because the linear merges minimize disk accesses.

Fancy Sorting: Quick Sort



Algorithm

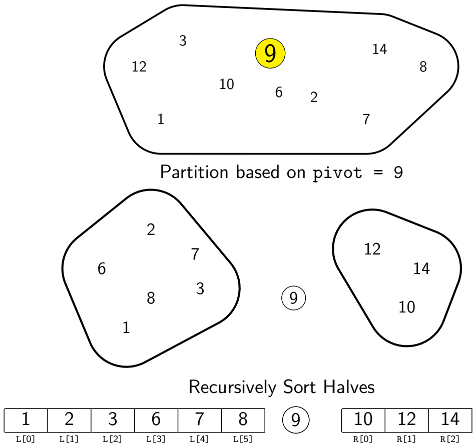
```
sort(A) {
   if (A.length < 2) {
      return A;
   }

   pivot = choosePivot(A);
   left = sort(getLess(A, pivot));
   right = sort(getGreater(A, pivot
        ));
   return left + pivot + right;
}
```

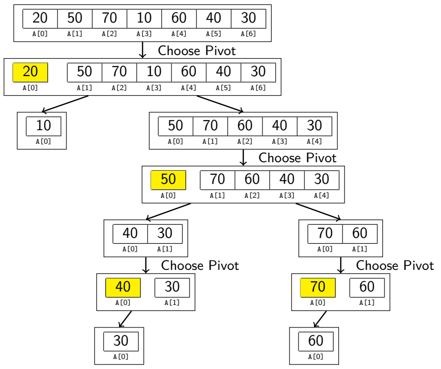
Runtime and Analysis

- Best Case?
- Average Case?
- Worst Case?
- In-Place?
- Stable?

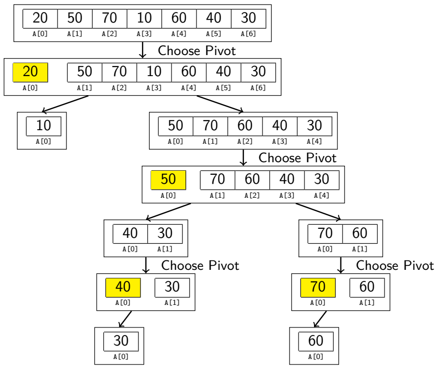
Fancy Sorting: Quick Sort



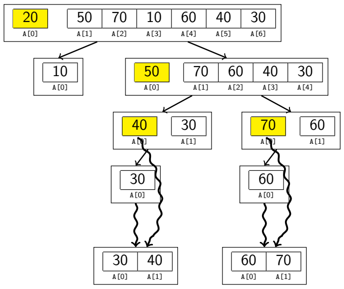
Fancy Sorting: Quick Sort



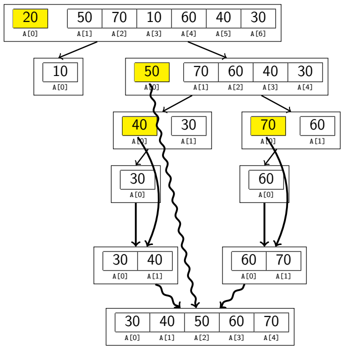
Fancy Sorting: Quick Sort



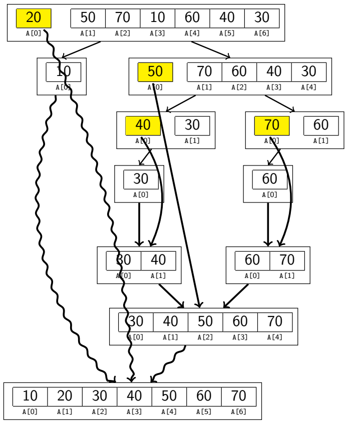
Fancy Sorting: Quick Sort



Fancy Sorting: Quick Sort



Fancy Sorting: Quick Sort



Fancy Sorting: Quick Sort

We now have the general idea of Quick Sort, but there are some remaining questions:

How do we choose the pivot?

How do we partition the array?

QuickSort: Good and Bad Pivots

Best Pivot?

If we had our choice of pivots, which one would we choose?

Median

The median will halve the problem each recursive call.

Worst Pivot?

If an adversary chose our pivot (to make the algorithm take as long as possible), which one would they choose?

Minimum or **Maximum**

This will decrease the problem size by only **one** each recursive call.

Pivot Strategies

There are several "standard" strategies to choose a pivot:

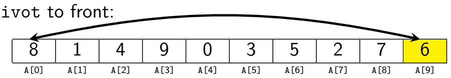
1. Choose the first/last element of the array
   - Very fast!
   - Bad, because real-world data is usually "mostly sorted"
2. Random choice
   - Generation can be slow
   - Good, because there's no easy worst case
3. Median of first, middle, and last elements
   - Works well in practice

Partitioning with Median-Of-Three Pivot

Choose a pivot as the median of `lo`, `mid`, and `hi`:

8	1	4	9	0	3	5	2	7	6
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

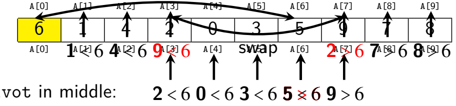
Move `pivot` to front:



8	1	4	9	0	3	5	2	7	6
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

Move < `pivot` to the front and > `pivot` to the end:

6	1	4	9	0	3	5	2	7	8
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]



Put `pivot` in middle:

5	1	4	2	0	3	6	9	7	8
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A[9]

QuickSort Analysis

Best Case

The best case is that the pivot is always the **median**. Then, we get two recursive calls each of size $n/2$.

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

So, the best case behavior is $\mathcal{O}(n\lg(n))$.

Worst Case

The worst case is that the pivot is always the **minimum** or the **maximum**. Then, we get one recursive call of size $n-1$.

$$T(n) = \begin{cases} T(n-1)+n & \text{if } n > 1 \\ 1 & \text{otherwise} \end{cases}$$

So, the worst case behavior is $\mathcal{O}(n^2)$.

Average Case

With a random pivot, on average we get $\mathcal{O}(n\lg(n))$ behavior.

QuickSort Cutoff

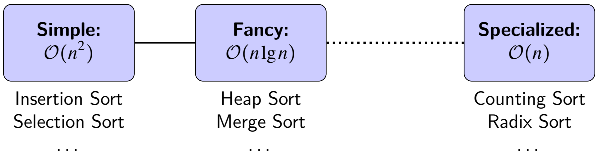
For small n, the recursion is a waste. The constants on quick/merge sort are higher than the ones on insertion/selection sort **for small** n.

The solution is to switch to a different algorithm for small sub-problems. For sorting, $n < 10$ is a good choice.

For example:

```
void quicksort(int[] arr, int lo, int hi) {
if (hi lo < CUTOFF) {
    insertionSort(arr,lo,hi);
}
else {
   ...
}
```

Where We Are



Simple: $\mathcal{O}(n^2)$

Insertion Sort
Selection Sort
...

Fancy: $\mathcal{O}(n\lg n)$

Heap Sort
Merge Sort
...

Specialized: $\mathcal{O}(n)$

Counting Sort
Radix Sort
...

We've discussed some sorting methods

They all happened to be $\Omega(n\lg n)$. Can we do better?