# Algorithm Efficiency and Sorting

# How to Compare Different Problems and Solutions

- Two different problems
  - Which is harder/more complex?
- Two different solutions to the same problem
  - Which is better?
- Questions:
  - How can we compare different problems and solutions?
  - What does it mean to say that one problem or solution is more simpler or more complex than another?

# Possible Solutions

- Idea: Code the solutions and compare them
  - Issues: machine, implementation, design, compiler, test cases, ...
- Better idea: Come up with a *machine- and implementation-independent* representation
  - # of steps
  - Time to do each step
- Use this representation to compare problems and solutions

# Example: Traversing a Linked List

```
Node curr = head;                        // time: c1
while(curr != null) {                    // time: c2
    System.out.println(curr.getItem());
    curr=curr.getNext();                 // time: c3
}
```

- Given n elements in the list, total time =

$$
\begin{aligned}
& 1 \times c_1 + (n+1) \times c_2 + n \times c_3 \\
& = n \times (c_2 + c_3) + c_2 + 1 \\
& = n \times d_1 + d_2 \\
& \propto n
\end{aligned}
$$

# Example: Nested Loops

```
for(i = 0; i < n; i++) {
    for(j = 0; j < n; j++) {
        System.out.println(i*j);    // time: c
    }
}
```

- Total time = $n \times n \times c$

$$\propto n^2$$

# Example: Nested Loops II

```
for(i = 0; i < n; i++) {
    for(j = 0; j < i; j++) {
        System.out.println(i*j);    // time: c
    }
}
```

- Total time = $\sum_{i=1}^{n} i \times c = c \sum_{i=1}^{n} i$

$$= c \times n \times (n-1)/2$$

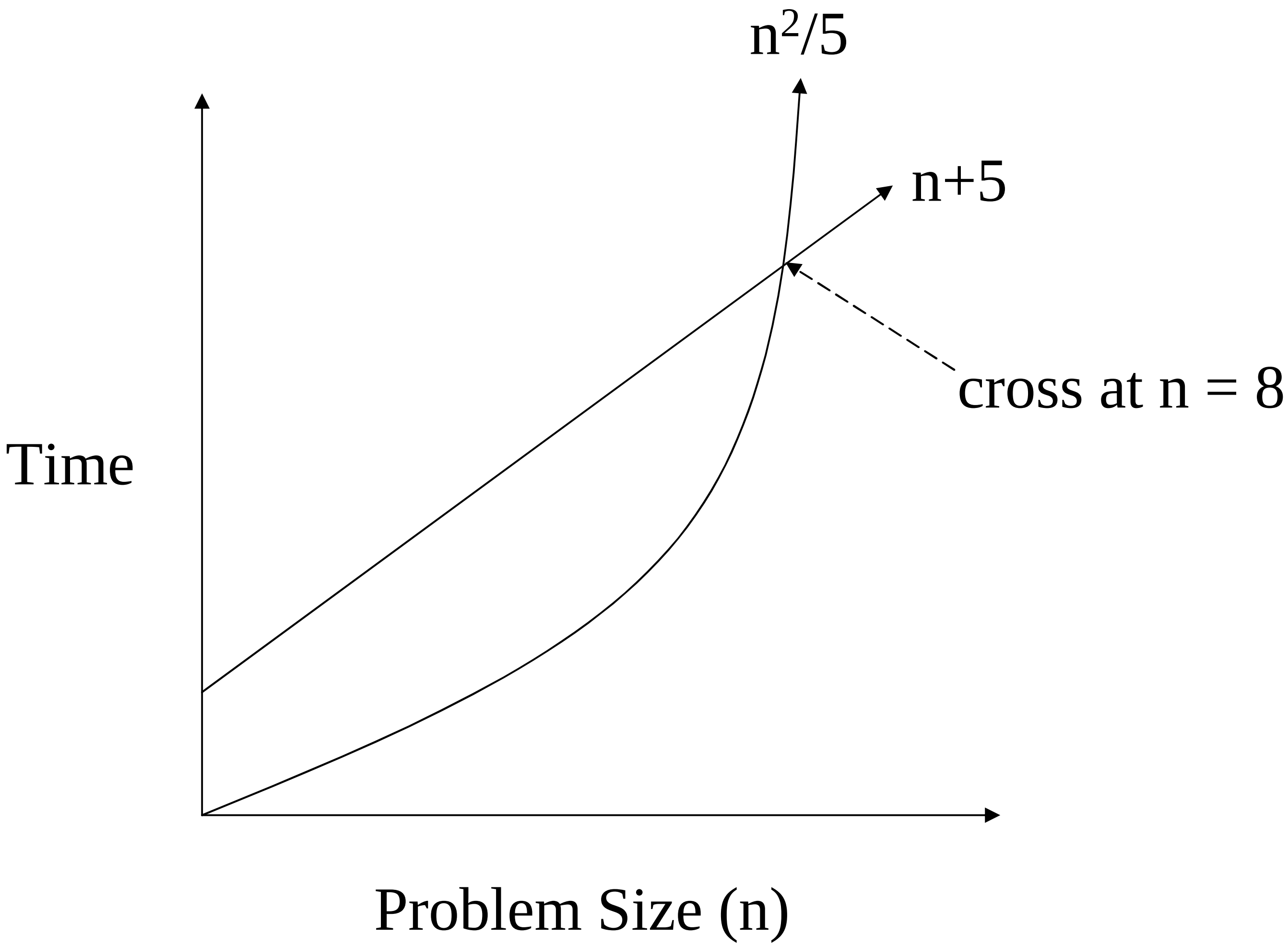
$$= d \times (n^2 - n)$$

$$\propto n^2 - n$$

# Results

- Which algorithm is better?
  - Algorithm A takes $n^2 - 37$ time units
  - Algorithm B takes n+45 time units
- Key Question: What happens as n gets large?
- Why?
  - Because for small n you can use any algorithm
  - Efficiency usually only matters for large n
- Answer: Algorithm B is better for large n
- Unless the constants are large enough
  - $n^2$
  - n + 1000000000000

# Graphically



# Big O notation: O(n)

- An algorithm g(n) is proportional to f(n) if $g(n)=c_1f(n)+c_2$
  - where $c_1 \neq 0$
- If an algorithm takes time proportional to f(n), we say the algorithm is **order f(n)**, or **O(f(n))**
- Examples
  - $n+5$ is $O(n)$
  - $(n^2 + 3)/2$ is $O(n^2)$
  - $5n^2+2n/17$ is $O(n^2 + n)$

# Exact Definition of O(f(n))

- An algorithm A is O(f(n))
- IF there exists k and $n_0$
- SUCH THAT A takes at most $k \times f(n)$ time units
- To solve a problem of size $n \geq n_0$

- Examples:
- $n/5 = O(n)$: $k = 5$, $n_0 = 1$
- $3n^2+7 = O(n^2)$: $k = 4$, $n_0 = 3$

- In general, toss out constants and lower-order terms, and $O(f(n)) + O(g(n)) = O(f(n) + g(n))$

# Relationships between orders

- $O(1) < O(\log_2 n)$
- $O(\log_2 n) < O(n)$
- $O(n) < O(n\log_2 n)$
- $O(n\log_2 n) < O(n^2)$
- $O(n^2) < O(n^3)$
- $O(n^x) < O(x^n)$, for all x and n

# Intuitive Understanding of Orders

- O(1) – Constant function, independent of problem size
  - Example: Finding the first element of a list
- $O(\log_2 n)$ – Problem complexity increases slowly as the problem size increases.
  - Squaring the problem size only doubles the time.
  - Characteristic: Solve a problem by splitting into constant fractions of the problem (e.g., throw away ½ at each step)
  - Example: Binary Search.
- O(n) – Problem complexity increases linearly with the size of the problem
  - Example: counting the elements in a list.

# Intuitive Understanding of Orders

- $O(n\log_2 n)$ – Problem complexity increases a little faster than n
  - Characteristic: Divide problem into subproblems that are solved the same way.
  - Example: mergesort
- $O(n^2)$ – Problem complexity increases fairly fast, but still manageable
  - Characteristic: Two nested loops of size n
  - Example: Introducting everyone to everyone else, in pairs
- $O(2^n)$ – Problem complexity increases very fast
  - Generally unmanageable for any meaningful n
  - Example: Find all subsets of a set of n elements

# Search Algorithms

- Linear Search is O(n)
  - Look at each element in the list, in turn, to see if it is the one you are looking for
  - Average case n/2, worst case n
- Binary Search is $O(\log_2 n)$
  - Look at the middle element m. If x < m, repeat in the first half of the list, otherwise repeat in the second half
  - Throw away half of the list each time
  - Requires that the list be in sorted order
    - Sorting takes $O(n\log_2 n)$
- Which is more efficient?

# Sorting

# Selection Sort

- For each element i in the list
  - Find the smallest element j in the rest of the list
  - Swap i and j
- What is the efficiency of Selection sort?
- The for loop has n steps (1 per element of the list)
- Finding the smallest element is a linear search that takes n/4 steps on average (why?)
- The loops are nested: $n \times n/2$ on average: $O(n^2)$

# Bubble sort

- Basic idea: run through the array, exchanging values that are out of order
  - May have to make multiple "passes" through the array
  - Eventually, we will have exchanged all out-of-order values, and the list will be sorted
  - Easy to code!
- Unlike selection sort, bubble sort doesn't have an outer loop that runs once for each item in the array
- Bubble sort works well with either linked lists or arrays

# Bubble sort: code

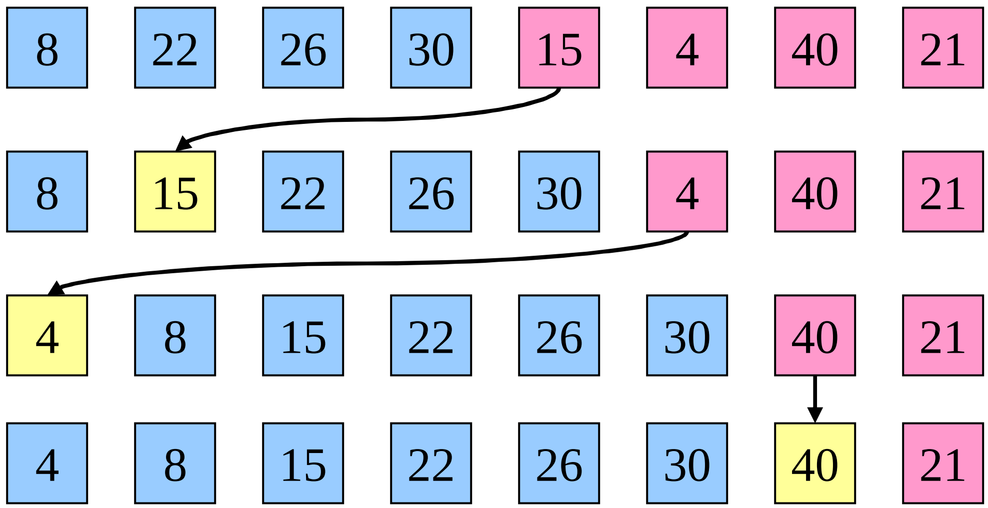
```
boolean done = false;
while(!done) {
  done = true;
  for (j = 0; j < length -1; j++)
{
    if (arr[j] > arr[j+1]) {
      temp = arr[j];
      arr[j] = arr[j+1];
      arr[j+1] = temp;
      done = false;
    }
  }
}
```

- Code is very short and simple
- Will it ever finish?
  - Keeps going as long as at least one swap was made
  - How do we know it’ll eventually end?
- Guaranteed to finish: finite number of swaps possible
  - Small elements “bubble” up to the front of the array
  - Outer loop runs at most nItems-1 times
- Generally not a good sort
  - OK if a few items slightly out of order

# Bubble sort: running time

- How long does bubble sort take to run?
  - Outer loop can execute a maximum of nItems-1 times
  - Inner loop can execute a maximum of nItems-1 times
- Answer: $O(n^2)$
  - Best case time could be much faster
  - Array nearly sorted would run very quickly with bubble sort
- Beginning to see a pattern: sorts seem to take time proportional to $n^2$
  - Is there any way to do better?
  - Let’s check out insertion sort

# What is insertion sort?




- Insertion sort: place the next element in the unsorted list where it “should” go in the sorted list
  - Other elements may need to shift to make room
  - May be best to do this with a linked list…

# Pseudocode for insertion sort

```
while (unsorted list not empty) {
  pop item off unsorted list
  for (cur = sorted.first;
       cur is not last && cur.value < item.value;
       cur = cur.next) {
    ;
  if (cur.value < item.value) {
    insert item after cur // last on list
  } else {
    insert item before cur
  }
}
```

# How fast is insertion sort?

- Insertion sort has two nested loops
  - Outer loop runs once for each element in the original unsorted loop
  - Inner loop runs through sorted list to find the right insertion point
    - Average time: 1/2 of list length
- The timing is similar to selection sort: $O(n^2)$
- Can we improve this time?
  - Inner loop has to find element just past the one we want to insert
  - We know of a way to this in O(log n) time: binary search!
    - Requires arrays, but insertion sort works best on linked lists…
    - Maybe there's hope for faster sorting

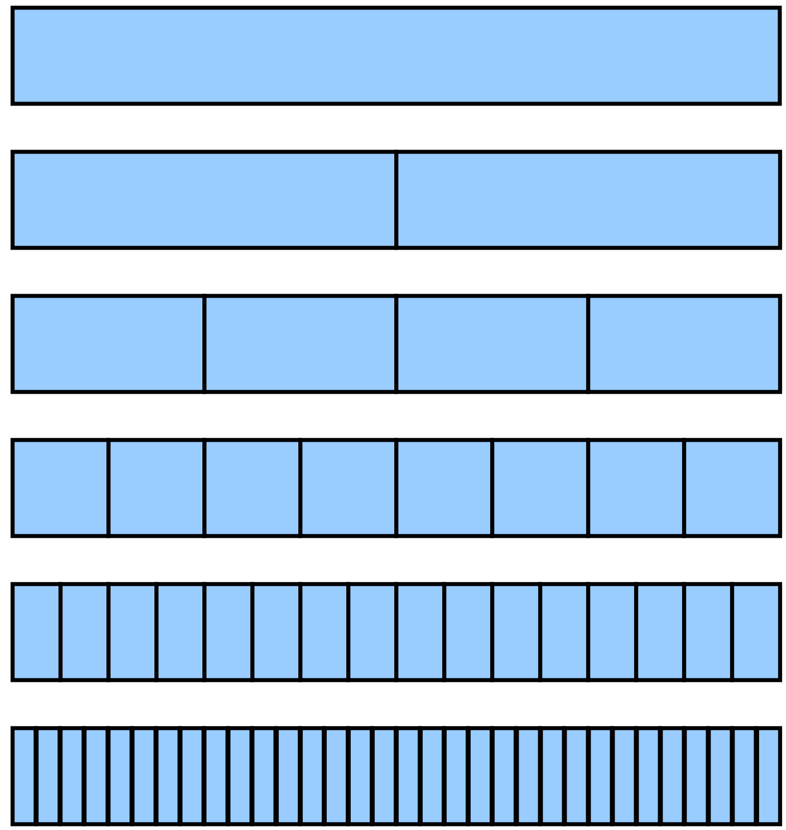
# How can we write faster sorting algorithms?

- Many common sorts consist of nested loops ($O(n^2)$)
  - Outer loop runs once per element to be sorted
  - Inner loop runs once per element that hasn't yet been sorted
    - Averages half of the set to be sorted
  - Examples
    - Selection sort
    - Insertion sort
    - Bubble sort
- Alternative: recursive sorting
  - Divide set to be sorted into two pieces
  - Sort each piece recursively
  - Examples
    - Mergesort
    - Quicksort

# Sorting by merging: mergesort

1. Break the data into two equal halves
2. Sort the halves
3. Merge the two sorted lists

- Merge takes O(n) time
  - 1 compare and insert per item
- How do we sort the halves?
  - Recursively
- How many levels of splits do we have?
  - We have O(log n) levels!
  - Each level takes time O(n)
- O(n log n)!



# Mergesort: the algorithm

```
void mergesort (int arr[], int sz) {
  int half = sz/2;
  int *arr2;
  int k1, k2, j;
  if (sz == 1) {
    return;
  }
  arr2 = (int *)malloc(sizeof (int) * sz);
  bcopy (arr, arr2, sz*sizeof(int));
  mergesort (arr2, half);
  mergesort (arr2+half, sz-half);
  for (j=0, k1=0, k2=half; j < sz; j++) {
    if ((k1 < half) && ((k2 >= sz) || (arr2[k1] < arr2[k2]))) {
      arr[j] = arr2[k1++];
    } else {
      arr[j] = arr2[k2++];
    }
  }
  free (arr2);
}
```

Any array of size 1 is sorted!

Make a copy of the data to sort

Recursively sort each half

Merge the two halves

Use the item from first half if any left and
- There are no more in the second half or
- The first half item is smaller

Free the duplicate array

# How well does mergesort work?

- Code runs in O(n log n)
  - O(n) for each “level”
  - O(log n) levels
- Depending on the constant, it may be faster to sort small arrays (1–10 elements or so) using an $n^2$ sort

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| 53 | 53 | 53 | 53 | 53 | 14 | 2 | 2 |
| 14 | 14 | 14 | 14 | 14 | 27 | 14 | 7 |
| 27 | 27 | 27 | 27 | 27 | 53 | 27 | 11 |
| 2 | 2 | 2 | 2 | 2 | 2 | 31 | 14 |
| 31 | 31 | 31 | 31 | 31 | 31 | 53 | 27 |
| 85 | 85 | 85 | 85 | 85 | 85 | 85 | 30 |
| 30 | 30 | 30 | 30 | 30 | 11 | 7 | 31 |
| 11 | 11 | 11 | 11 | 11 | 30 | 11 | 39 |
| 67 | 67 | 67 | 67 | 67 | 67 | 30 | 50 |
| 50 | 50 | 50 | 50 | 50 | 7 | 39 | 53 |
| 7 | 7 | 7 | 7 | 7 | 39 | 50 | 67 |
| 39 | 39 | 39 | 39 | 39 | 50 | 67 | 85 |

# Problems with mergesort

- Mergesort requires two arrays
  - Second array dynamically allocated (in C)
  - May be allocated on stack in C++
    int arr2[sz];
  - This can take up too much space for large arrays!
- Mergesort is recursive
- These two things combined can be real trouble
  - Mergesort can have log *n* recursive calls
  - Each call requires O(*n*) space to be allocated
- Can we eliminate this need for memory?

# Solution: mergesort “in place”

- Mergesort builds up “runs” of correctly ordered items and then merges them
- Do this “in place” using linked lists
  - Eliminates extra allocation
  - Eliminates need for recursion (!)
- Keep two lists, each consisting of runs of 1 or more elements in sorted order
  - Combine the runs at the head of the lists into a single (larger) run
  - Place the run at the back of one of the lists
  - Repeat until you’re done

# Mergesort “in place” in action

| | | | | | | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 53 | 2 | 44 | 85 | 11 | 67 | 7 | 39 | 14 | 53 | 87 | 11 | 50 | 67 | 2 | 14 | 44 | 53 | 80 | 85 | 87 |

| | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 14 | 87 | 80 | 50 | 29 | 72 | 95 | 2 | 44 | 80 | 85 | 7 | 29 | 39 | 72 | 95 |

- Boxes with same color are in a single “run”
  - Specific color has no other meaning
- Runs get larger as the algorithm runs
  - Eventually, entire set is in one run!
- Algorithm works well with linked lists
  - No need to allocate extra arrays for merging!

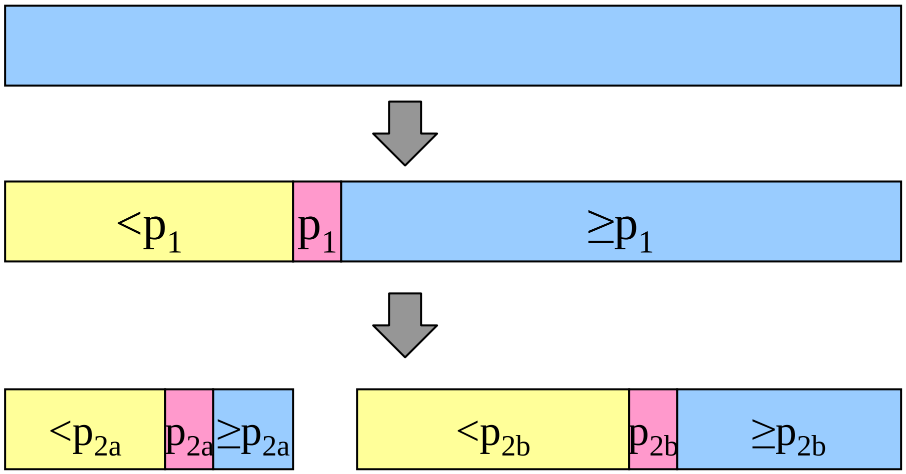
# Benefits of mergesort "in place"

- Algorithm may complete faster than standard mergesort
  - Requires fewer iterations if array is nearly sorted (lots of long runs)
  - Even small amounts of order make things faster
- No additional memory need be allocated
- No recursion!
  - Recursion can be messy if large arrays are involved
- Works well with linked lists
  - Standard mergesort is tougher with linked lists: need to find the "middle" element in a list
- May be less copying: simply rearrange lists

# Quicksort: another recursive sort

- "Standard" mergesort requires too much memory
  - Extra array for merging
- Alternative: use quicksort
- Basic idea: partition array into two (possibly unequal) halves using a *pivot* element
  - Left half is all less than pivot
  - Right half is all greater than pivot
- Recursively continue to partition each half until array is sorted
  - Elements in a partition may move relative to one another during recursive calls
  - Elements can't switch partitions during recursion

# How quicksort works




- Pick a pivot element
- Divide the array to be sorted into two halves
  - Less than pivot
  - Greater than pivot
  - Need not be equal size!
- Recursively sort each half
  - Recursion ends when array is of size 1
  - Recursion may instead end when array is "small": sort using traditional $O(n^2)$ sort
- How is pivot picked?
- What does algorithm look like?

# Quicksort: pseudocode

```
quicksort (int theArray[], int nElem)
{
 if (nElem <= 1)  // We're done
    return;
  Choose a pivot item p from theArray[]
  Partition the items of theArray about p
    Items less than p precede it
    Items greater than p follow it
    p is placed at index pIndex
  // Sort the items less than p
  quicksort (theArray, pIndex);
  // Sort the items greater than p
  quicksort (theArray+pIndex+1, nElem-(pIndex+1));
}
```

Key question: how do we pick a "good" pivot (and what makes a good pivot in the first place)?

# Picking a pivot

- Ideally, a pivot should divide the array in half
  - How can we pick the middle element?
- Solution 1: look for a "good" value
  - Halfway between max and min?
  - This is slow, but can get a good value!
  - May be too slow…
- Solution 2: pick the first element in the array
  - Very fast!
  - Can result in slow behavior if we're unlucky
- Most implementations use method 2

# Quicksort: code

```
quicksort (int theArray[ ], int nElem)
{
  int pivotElem, cur, tmp;
  int endS1 = 0;
  if (nElem <= 1)  return;
  pivotElem = theArray[0];
  for (cur = 1; cur < nElem; cur++) {
    if (theArray[cur] < pivotElem) {
       tmp = theArray[++endS1];
       theArray[endS1] = theArray[cur]);
       theArray[cur] = tmp;
    }
  }
  theArray[0] = theArray[endS1];
  theArray[endS1] = pivotElem;
  quicksort (theArray, endS1); // Sort the two parts of the array
  quicksort (theArray+endS1+1, nElem-(endS1+1));
}
```

# How fast is quicksort?

- Average case for quicksort: pivot splits array into (nearly) equal halves
  - If this is true, we need O(log $n$) "levels" as for mergesort
  - Total running time is then O($n$ log $n$)
- What about the worst case?
  - Pick the minimum (or maximum) element for the pivot
  - $S_1$ (or $S_2$) is empty at each level
  - This reduces partition size by 1 at each level, requiring $n$-1 levels
  - Running time in the worst case is O($n^2$)!
- For average case, quicksort is an excellent choice
  - Data arranged randomly when sort is called
  - May be able to ensure average case by picking the pivot intelligently
  - No extra array necessary!

# Radix Sort: O(n) (sort of)

- Equal length strings
- Group string according to last letter
- Merge groups in order of last letter
- Repeat with next-to-last letter, etc.
- Let's discuss how to do this
- Time: O(nd)
  - If d is constant (16-bit integers, for example), then radix sort takes O(n)