Sorting Algo

Sorting involves arranging a collection of items in order

- based on some property of the item (e.g. key)
- using an ordering relation on that property

Why is sorting useful?

- speeds up subsequent searching (binary searching need a sorted list)
- arranges data in a human-useful way (e.g. list of students in a tutor class, ordered by family-name or id)
- provides intermediate step for advanced algorithms (e.g. duplicate detection/removal, many DBMS operations)

Properties of sorting algorithms: *stable*, *adaptive*, *in-place*

- ***Stable***: if two items have duplicate keys, their relative order would not change in the sorted array.
  - let $x = a[i]$, $y = a[j]$, $key(x) == key(y)$, if x precedes y in a, then x precedes y in a'
  - This is particularly useful to repeated sorting different keys for an array.
- ***Adaptive***: whether it takes advantage of existing order in its input
  - behaviour/performance of algorithm affected by data values
  - i.e. best/average/worst case performance differs
  - e.g. bubble sort
- ***In-place sorting***: no additional storage space is needed to perform sorting.
  - Accordingly, ***out-place sorting*** needs to create a copy of array to create a new array.
- ***Comparison-based sorting***: A comparison sort is a type of sorting algorithm that only reads the list elements through a single abstract **comparison** operation (often a "less than or equal to" operator or

a three-way comparison) that determines which of two elements should occur first in the final sorted list.

Why stable is important:

> Suppose we have a list of first and last names. We are asked to sort "by last name, then by first". We could first sort (stable or unstable) by the first name, then stable sort by the last name. After these sorts, the list is primarily sorted by the last name. However, where last names are the same, the first names are sorted.

Some other properties:

Two major classes of sorting algo: $O(n^2)$, $O(n \log n)$

- $O(n^2)$ are acceptable if n is small (hundreds)

Basic framework for Sorting:

```
// we deal with generic Items
typedef int Item;

// abstractions to hide details of Items
#define key(A) (A)
#define less(A,B) (key(A) < key(B))
#define swap(A,B) {Item t; t = A; A = B; B = t;}
#define swil(A,B) {if (less(A,B)) swap(A,B);}
// swil = SWap If Less

// Sorts a slice of an array of Items
void sort(Item a[], int lo, int hi);

// Check for sortedness (to validate functions)
```

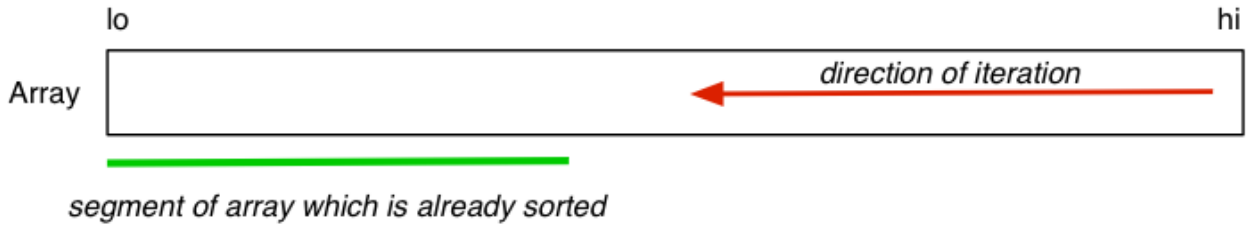
```
int isSorted(Item a[], int lo, int hi);
```

Two key sorting abstractions:

- `Compare` : whether item `v` is less than `w`
- `Swap` : swap item `v` and `w` in array `a[]`

Elementary Sorting Algo

To describe simple sorting, we use diagrams like:



In these algorithms:

- a ***segment*** of the array is already sorted, which we do not need to consider any more
- each ***iteration*** makes more of the array sorted

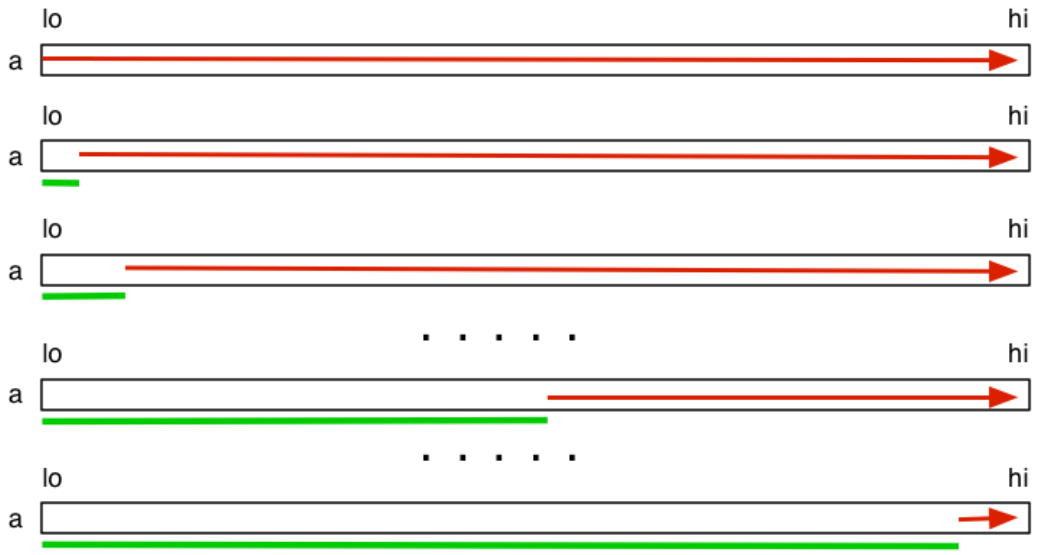
Performance of elementary: $O(n^k)$ where $k > 1$

Selection Sort: Simple non-adaptive method

Basic Idea:
For every iteration:

1. find the smallest element in the *unsorted segment*, put it into the last slot of the *sorted segment*

2. repeat until all elements are in the sorted segment



Performance: $O(n^2)$, not adaptive

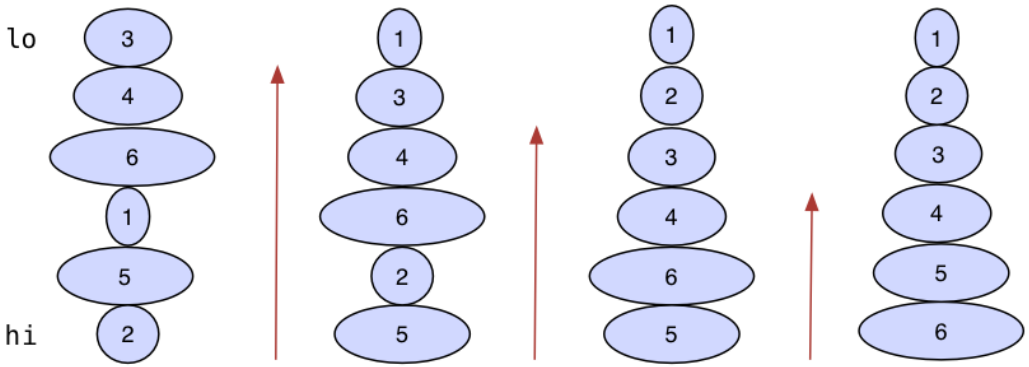
```
void selectionSort(int a[], int lo, int hi)
{
   int i, j, min;
   for (i = lo; i < hi-1; i++) {
      min = i;
      for (j = i+1; j <= hi; j++) {
         if (less(a[j],a[min])) min = j;
      }
      swap(a[i], a[min]);
   }
}
```

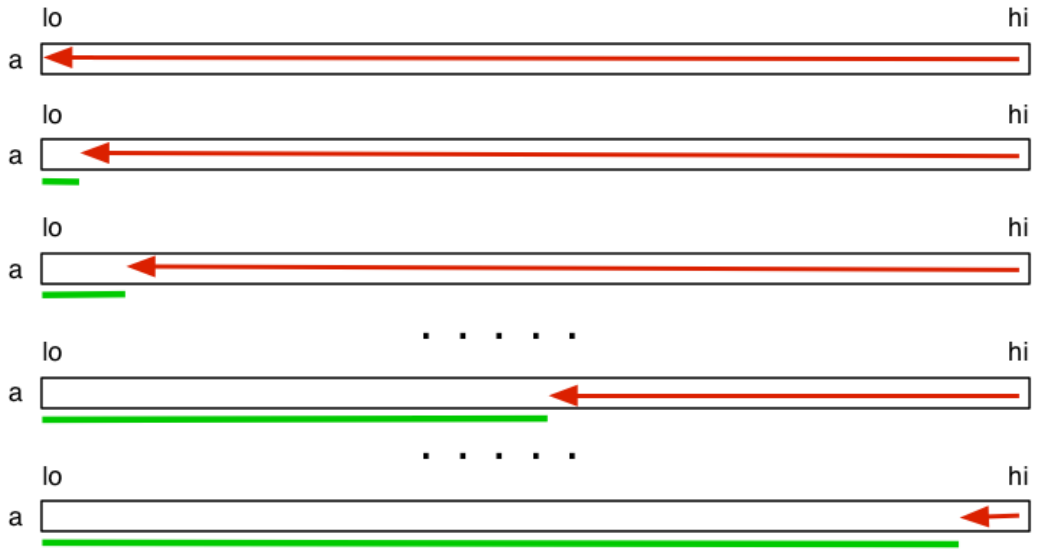
Bubble Sort: Simple adaptive method

Simple adaptive method:
For every iteration:

- elements move until they meet a smaller element, eventually `i` th smallest element moves to `i` th position
  - those small elements are in the sorted segment
- repeat until there are no swaps during one iteration (already completely sorted)

This idea is basically same as the insertion sort (move smaller elements to the sorted segment), but the advantage is that bubble sort could jump some already sorted elements.





Performance:

- best case: $O(n)$, the array is already sorted
- worst case: $O(n^2)$, the array is reverse sorted

Insertion Sort: Simple adaptive method

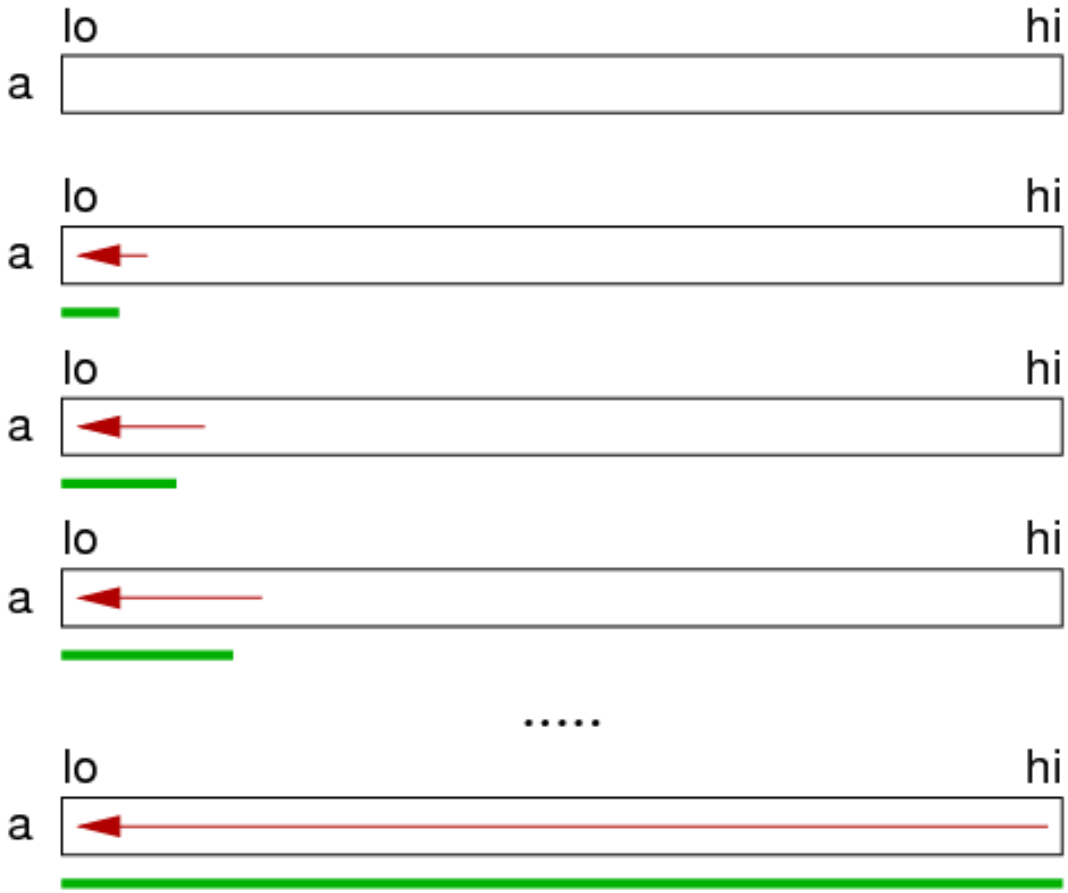
Basic ideas:

- First, take first element and treat as sorted segment (length 1).
- Then, for every iteration:
  - insert the first element in the *unsorted segment* into the correct position in the *sorted segment*
  - repeat until whole array is sorted

Steps:
Iterate from `0` to `N-1`, totally `N` runs

1. In iteration `i`, swap `a[i]` with each larger entry to its left
2. Continually compare and swap, until it is smaller than its left
3. Move `i` to `i+1`, repeat

Insertion sort is like the combination of reverse bubble sort and selection sort, where you use bubble in the sorted segment.



Performance:

- Key operations: For a randomly-ordered array with distinct keys, insertion sort uses $\frac{N^2}{4}$ compares

and $\frac{N^2}{4}$ exchanges on average

- best case: $O(n)$, the list is already sorted
- worst case: $O(n^2)$, the list is reverse sorted
- Note: The advantage that insertion sort has over bubble sort is that, when the list is mostly sorted, bubble sort still need to ***swap*** many times, while insertion sort can jump most of those compares and ***insert*** directly.

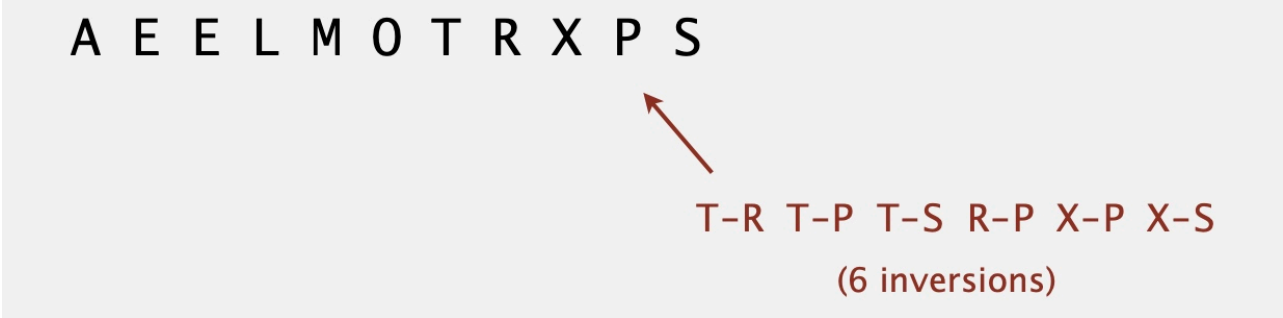
Insertion sort:

- based on exchanges that only involve adjacent items
- already improved above by using moves rather than swaps
- "long distance" moves may be more efficient

```
void insertionSort(int a[], int lo, int hi)
{
   int i, j, val;
   for (i = lo+1; i <= hi; i++) {
      val = a[i];
      for (j = i; j > lo; j--) {
         if (val >= a[j-1]) break;
         a[j] = a[j-1];
      }
      a[j] = val;
   }
}
```

Inversions and Partially sorted Array

An ***inversion*** is a pair of keys that are out of order.



An array is ***partially sorted*** if the number of inversions is ≤ c N.

For partially-sorted arrays, insertion sort runs in linear time. The number of exchanges equals the number of inversions.

Shell Sort: Improving Insertion Sort

Shell sort is a highly efficient sorting algorithm and is based on insertion sort algorithm. This algorithm avoids large shifts as in case of insertion sort, if the smaller value is to the far right and has to be moved to the far left.

Basic idea:

- array is `h`-sorted if taking every `h`'th element yields a sorted array
  - A 1-sorted array is a completely sorted array.
- an `h`-sorted array is made up of `n/h` interleaved sorted arrays

For every iteration:

- Insertion sort `h`-sort array for progressively smaller `h`, ending with `1`-sorted.

The reason we use Insertion sort for every run:

1. For big `h`, subarray is small
2. For small `h`, subarray is nearly in order (require **adaptability**)
3. For every `h`, we need to ensure that the previous sorts remain in order. (require **stability**)

Performance:

- depends on the sequence of `h` values
- not yet been fully analysed
- from $O(n^{3/2})$ to $O(n^{4/3})$
    - Therefore, it is **very efficient for medium-sized arrays**, though not so useful for large-sized arrays

```
void shellSort(int a[], int lo, int hi)
{
   int hvals[8] = {701, 301, 132, 57, 23, 10, 4, 1};
   int g, h, start, i, j, val;

   for (g = 0; g < 8; g++) {
      h = hvals[g];
      start = lo + h;
      for (i = start; i < hi; i++) {
         val = a[i];
         for (j = i; j >= start && less(val,a[j-h]); j -= h)
            move(a, j, j-h);
         a[j] = val;
      }
   }
}
```

Summary of elementary sorting Algo

	#compares	**-**	**-**	**#swaps**	**-**	**-**	**#moves**	**-**	**-**
	min	avg	max	min	avg	max	min	avg	max
Selection sort	n^2	n^2	n^2	n	n	n	.	.	.
Bubble sort	n	n^2	n^2	0	n^2	n^2	.	.	.
Insertion sort	n	n^2	n^2	.	.	.	n	n^2	n^2
Shell sort	n	$n^{4/3}$	$n^{4/3}$	.	.	.	1	$n^{4/3}$	$n^{4/3}$

Advanced Sorting Algo

Advanced sorting algo can achieve better performance than $O(n^k)\ (k > 1)$, which is $O(n \log n)$.

Quick Sort

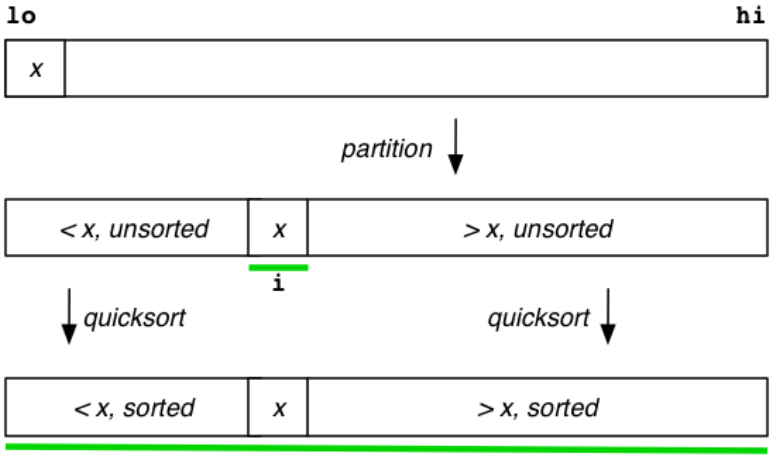
Quick sorting is not a stable or adaptive sorting algo.

Basic ideas:

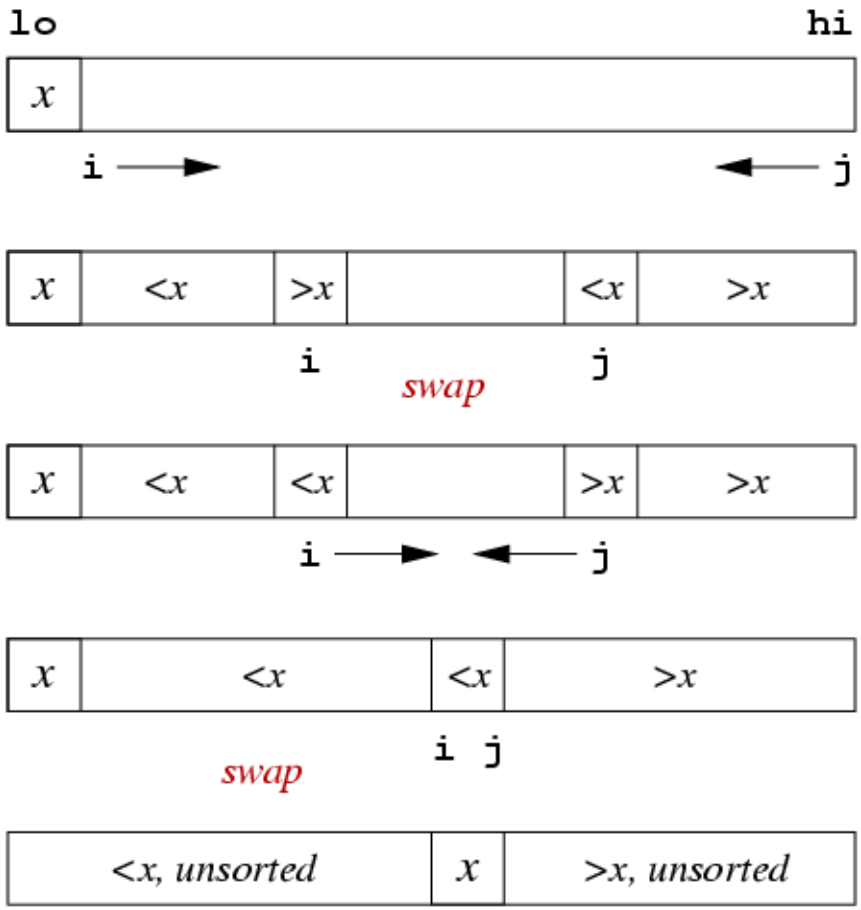
1. Pick a value `x` as ***pivot***
2. Use the pivot to ***partition*** the array into 2 parts (`<x` and `>x`)
3. Recursively sort each of the partitions

Vanilla Quick Sort

Vanilla Quick Sort is the most simple quick sort, where we pick the first value of the list as the pivot.



Partition is the most important part of quick sorting:




```
void quicksort(Item a[], int lo, int hi)
{
   int i; // index of pivot
   if (hi <= lo) return;
   i = partition(a, lo, hi);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}

int partition(Item a[], int lo, int hi)
{
```

```
    Item v = a[lo];  // pivot
    int  i = lo+1, j = hi;
    for (;;) {
        // let i point to the item greater than mid
        while (less(a[i], v) && i < j) i++;
        // let j point to the item smaller than mid
        while (less(v, a[j]) && j > i) j--;
        // if i and j meet, end
        if (i == j) break;
        swap(a, i, j);
    }
    j = less(a[i],v) ? i : i-1;
    swap(a, lo, j);
    return j;
}
```

Performance:

- Best case: $O(n \log n)$ comparisons
  - choice of pivot gives two equal-sized partitions
  - same happens at every recursive level
  - each "level" requires approx n comparisons
  - halving at each level ⇒ log2n levels
- Worst case: $O(n^2)$ comparisons
  - always choose lowest/highest value for pivot
  - partitions are size 1 and n-1
  - each "level" requires approx n comparisons
  - partitioning to 1 and n-1 ⇒ n levels

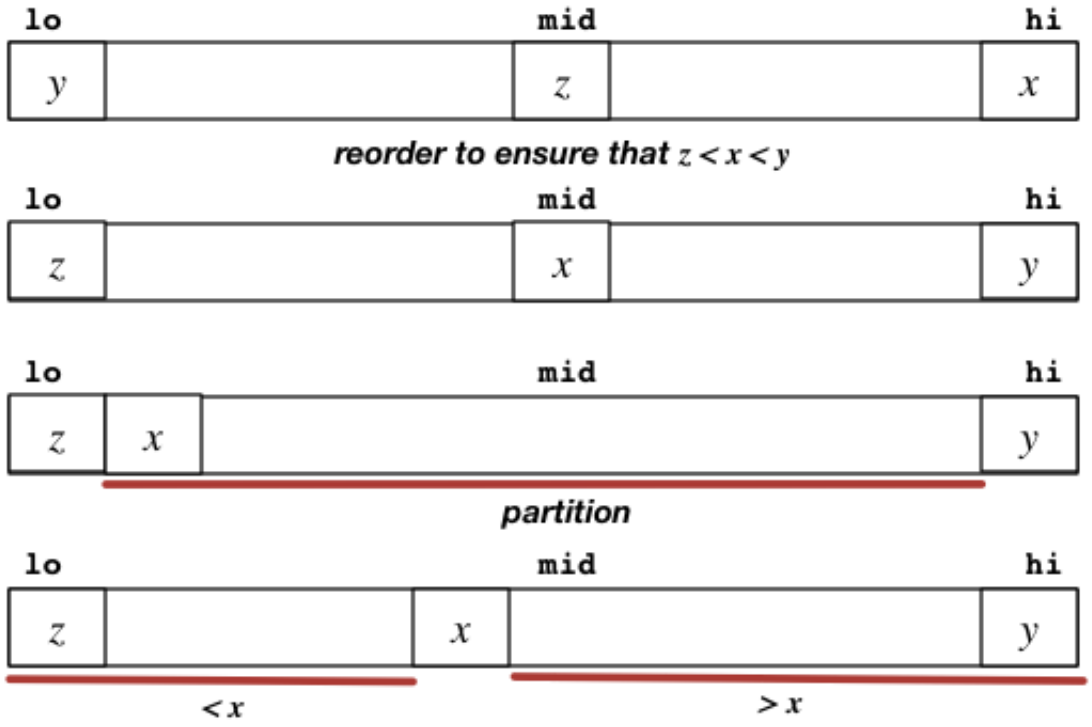
Improved Qucik Sort

Handle max/min pivot: Median-of-three

Choice of pivot can have significant effect: always choosing largest/smallest ⇒ worst case, which

So we need to try to find "intermediate" value.

One way is to choose pivot by median-of-three:




```
void medianOfThree(Item a[], int lo, int hi)
{
   int mid = (lo+hi)/2;
   if (less(a[mid],a[lo])) swap(a, lo, mid);
   if (less(a[hi],a[mid])) swap(a, mid, hi);
   if (less(a[mid],a[lo])) swap(a, lo, mid);
```

```
   // now, we have a[lo] < a[mid] < a[hi]
   // swap a[mid] to a[lo+1] to use as pivot
   swap(a, mid, lo+1); swap(a, lo, mid);
}

void quicksort(Item a[], int lo, int hi)
{
   int i;
   if (hi-lo < Threshhold) { ... return; }
   medianOfThree(a, lo, hi);
   i = partition(a, lo+1, hi-1);
   quicksort(a, lo, i-1);
   quicksort(a, i+1, hi);
}
```

Handle small partitions differently

Another source of inefficiency:

- pushing recursion down to very small partitions
- overhead in recursive function calls
- little benefit from partitioning when size < 5

  Solution: handle small partitions differently
- switch to ***insertion sort*** on small partitions, or
- don't sort yet; use post-quicksort insertion sort

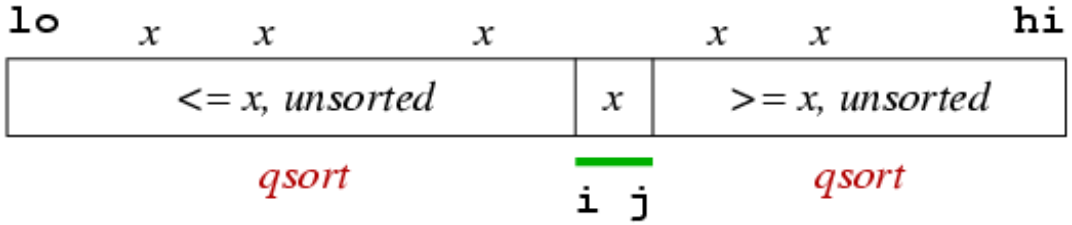
```
void quicksort(Item a[], int lo, int hi)
{
   int i;
   if (hi-lo < Threshhold) {
      insertionSort(a, lo, hi);
```

```
        return;
    }
    medianOfThree(a, lo, hi);
    i = partition(a, lo+1, hi-1);
    quicksort(a, lo, i-1);
    quicksort(a, i+1, hi);
}
```

Handle Duplicate keys: Three-way partitioning

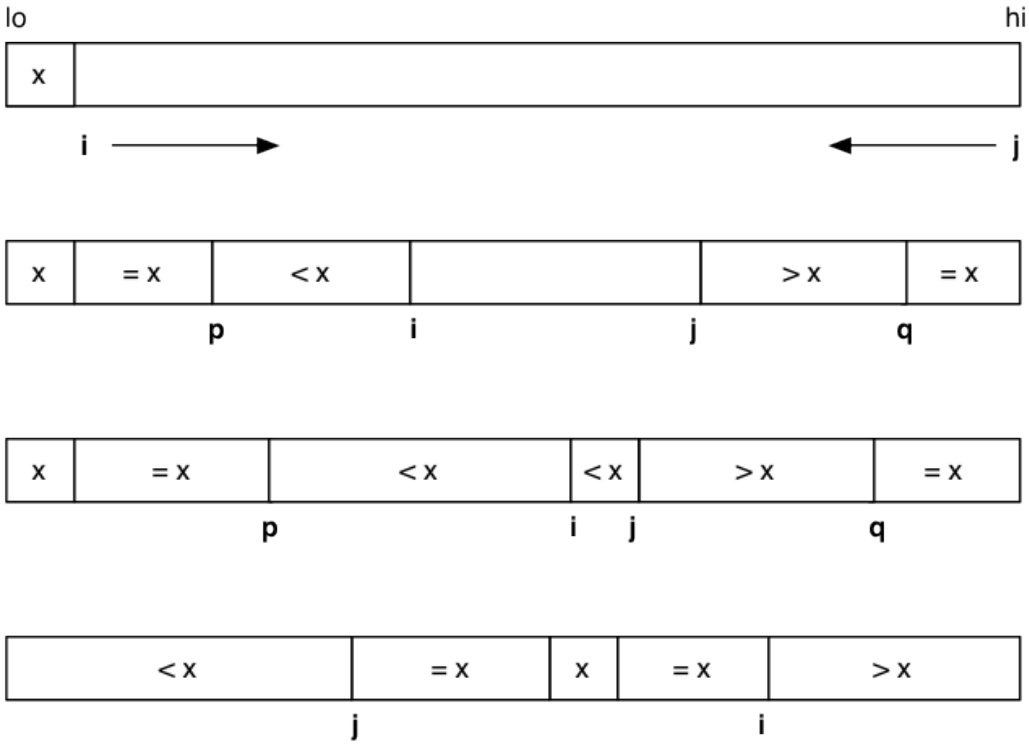
If the array contains many duplicate keys

- standard partitioning does not exploit this:



- can improve performance via ***three-way partitioning***: -w500

Bentley/McIlroy approach to three-way partition:



Non-recursive Quicksort

Quick sort can be implemented using an explicit stack:

```
void quicksortStack (Item a[], int lo, int hi)
{
   int i;  Stack s = newStack();
   StackPush(s,hi); StackPush(s,lo);
   while (!StackEmpty(s)) {
      lo = StackPop(s);
      hi = StackPop(s);
      if (hi > lo) {
         i = partition (a,lo,hi);
         StackPush(s,hi); StackPush(s,i+1);
```

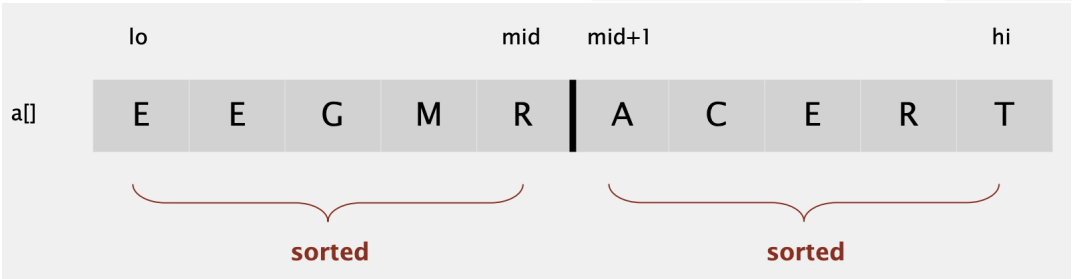
```
            StackPush(s,i-1); StackPush(s,lo);
        }
    }
}
```

Merge Sort

Basic idea: (like reverted Quicksort)

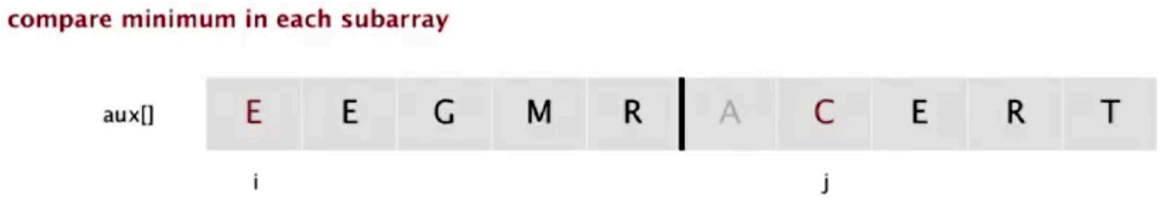
1. split the array into two equal-sized partitions until every partition is sorted
2. (recursively) sort each half
3. ***merge*** the two partitions back to original array

Merging is the critical step.
Example of a in-place merging:
Suppose we have 2 sorted subarrays: `a[lo] – a[mid]` and `a[mid+1] – a [hi]`



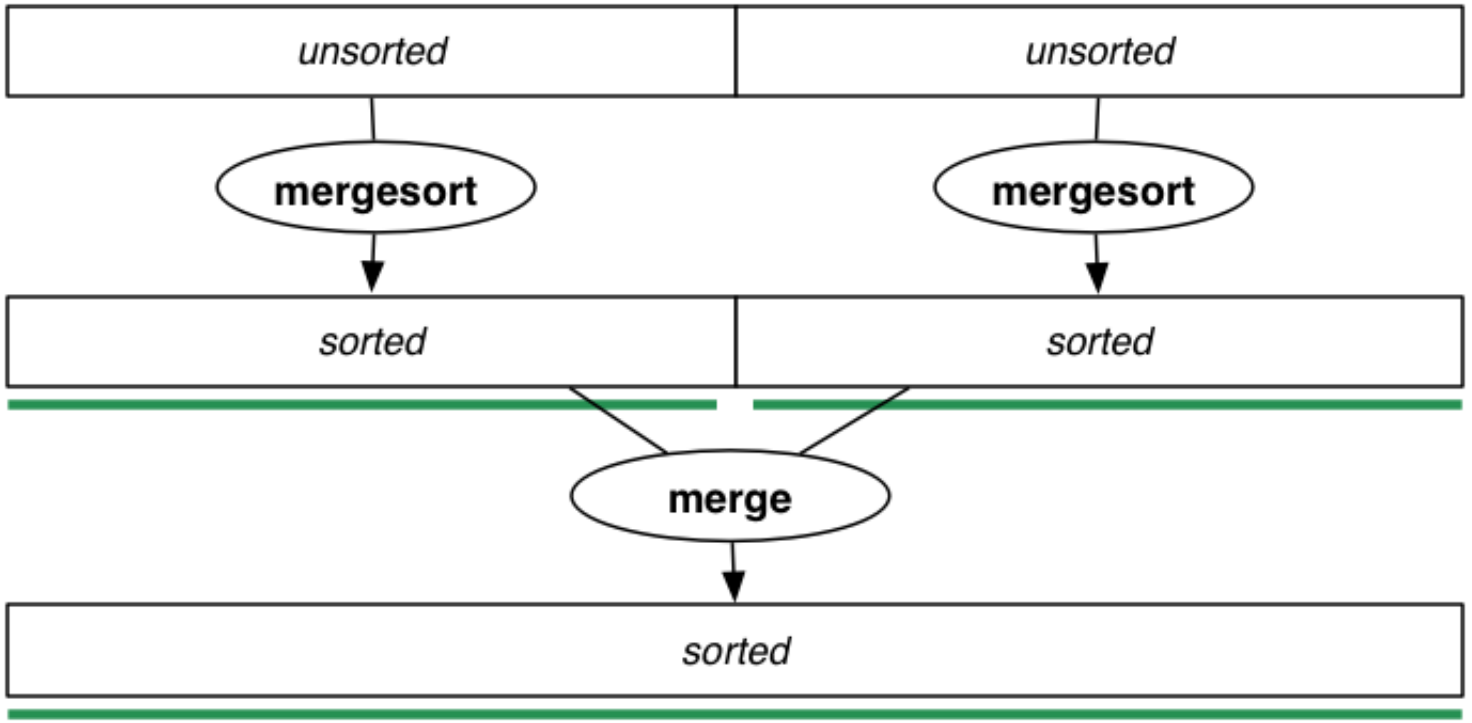
1. copy this array into one auxiliary array `aux`, and make pointers `i` and `j` to the smallest values of 2 subarrays

2. (recursively) copy the smaller of the two pointers, and increment the pointer



3. when one exhausted, copy the rest of the other

In mergesort, sorting is implemented by merging. They are equivalent.



In-place merging: Actually, we can just use two arrays to implement the merge sort to use less memory.

Best case: $O(N \log N)$ comparisons

- split array into equal-sized partitions
- same happens at every recursive level
- each "level" requires ≤ N comparisons
- halving at each level ⇒ log2N levels

Worst case: $O(N \log N)$ comparisons

- partitions are exactly interleaved
- need to compare all the way to end of partitions

Disadvantage over quicksort: need extra storage $O(N)$

```
void mergesort(Item a[], int lo, int hi)
{
    int mid = (lo+hi)/2; // mid point
    if (hi <= lo) return;
    mergesort(a, lo, mid);
    mergesort(a, mid+1, hi);
    merge(a, lo, mid, hi);
}

int nums[10] = {32,45,17,22,94,78,64,25,55,42};
mergesort(nums, 0, 9);

void merge(Item a[], int lo, int mid, int hi)
{
    int  i, j, k, nitems = hi-lo+1;
    // in-place merge just one additional array
    Item *tmp = malloc(nitems*sizeof(Item));

    i = lo; j = mid+1; k = 0;
    // scan both segments, copying to tmp
```

```
    while (i <= mid && j <= hi) {
        if (less(a[i],a[j]))
            tmp[k++] = a[i++];
        else
            tmp[k++] = a[j++];
    }
    // copy items from unfinished segment
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi) tmp[k++] = a[j++];

    //copy tmp back to main array
    for (i = lo, k = 0; i <= hi; i++, k++)
        a[i] = tmp[k];
    free(tmp);
}
```

Non-recursive Mergesort: Bottom-up mergesort ?

Non-recursive mergesort does not require a stack, because partition boundaries can be computed iteratively.

```
#define min(A,B) ((A < B) ? A : B)

// lo is the first index, hi is the last index
void mergesort(Item a[], int lo, int hi)
{
   int i, m; // m = length of runs
   int end; // end of 2nd run
   for (m = 1; m <= lo-hi; m = 2*m) {
      for (i = lo; i <= hi-m; i += 2*m) {
         end = min(i+2*m-1, hi);
         merge(a, i, i+m-1, end); // this still need a tmp array
```

```
        }
    }
}

void merge(Item a[], int lo, int mid, int hi)
{
    int  i, j, k, nitems = hi-lo+1;
    // in-place merge just one additional array
    Item *tmp = malloc(nitems*sizeof(Item));

    i = lo; j = mid+1; k = 0;
    // scan both segments, copying to tmp
    while (i <= mid && j <= hi) {
        if (less(a[i],a[j]))
            tmp[k++] = a[i++];
        else
            tmp[k++] = a[j++];
    }
    // copy items from unfinished segment
    while (i <= mid) tmp[k++] = a[i++];
    while (j <= hi) tmp[k++] = a[j++];

    //copy tmp back to main array
    for (i = lo, k = 0; i <= hi; i++, k++)
        a[i] = tmp[k];
    free(tmp);
}
```

Mergesort Variation

```
// lo is the first index, hi is the last index
void mergeSort(Item a[], int lo, int hi)
```

```
{
   int i;
   Item *aux = malloc((hi+1)*sizeof(Item));
   for (i = lo; i <= hi; i++) aux[i] = a[i];
   doMergeSort(a, aux, lo, hi);
   free(aux);
}

void doMergeSort(Item a[], Item b[], int lo, int hi)
{
    // if not need sort, return
    if (lo >= hi) return;
    // otherwise, merge sort
    // find the mid
    int mid = (lo+hi)/2;
    // mergesort two partitions
    doMergeSort(b, a, lo, mid);
    doMergeSort(b, a, mid+1, hi);
    // merge those sorted arrays
    merge(b+lo, mid-lo+1, b+mid+1, hi-mid, a+lo);
}

// what actually sorts array
// merge arrays a[] and b[] into c[]
// aN = size of a[], bN = size of b[]
void merge(Item a[], int aN, Item b[], int bN, Item c[]) {
    int i; // index into a[]
    int j; // index into b[]
    int k; // index into c[]
    for (i = j = k = 0; k < aN+bN; k++) {
        if (i == aN)
            c[k] = b[j++];
        else if (j == bN)
```

```
            c[k] = a[i++];
        else if (less(a[i],b[j]))
            c[k] = a[i++];
        else
            c[k] = b[j++];
    }
}
```

External Merge Sort

Previous sorts all assume ...

- efficient random access to Items
- which suggests that data is in arrays in memory
- which limits sortable data to what fits in memory

When the data is in disk files

- random access is inefficient (files are sequential access)
- but max data size is far less constrained

Because mergesort makes multiple sequential passes, it adapts well as a sorting approach for files.

External mergesort basic idea:

- have two files, A and B, which alternate as input/output
- scan input, sorting adjacent pairs, write to output
- scan input, merging pairs to sorted runs of length 4
- scan input, merging pairs to sorted runs of length 8
- repeat until entire file is sorted

How many iterations are needed?

- double scan length until ≥ file size (N Items)
- #iterations = $\log_2 N$


```
Input: stdin, containing N Items
Output: stream of Items on stdout

copy stdin file to A
runLength = 1
iter = 0
while (runLength < N) {
    if (iter % 2 == 0)
```

```
        inFile = A, outFile = B
    else
        inFile = B, outFile = A
    fileMerge(inFile, outFile, runLength, N)
    iter++;
    runLength *= 2;
}
copy outfile to stdout

// assumes N = 2^k for some integer k > 0
fileMerge(inFile, outFile, runLength, N)
{
   inf1 = open inFile for reading
   inf2 = open inFile for reading
   outf = open outFile for writing
   in1 = 0; in2 = runLength
   while (in1 < N) {
      seek to position in1 in inf1
      end1 = in1+runLength
      it1 = getItem(inf1)
      seek to position in2 in inf2
      end2 = in2+runLength
      it2 = getItem(inf2)
      while (in1 < end1 && in2 < end2) {
         if (less(it1,it2)) {
            write it1 to outf
            it1 = getItem(inf1); in1++
         }
         else {
            write it2 to outf
            it2 = getItem(inf2); in2++
         }
      }
```

```
        while (in1 < end1) {
            write it1 to outf
            it1 = getItem(inf1); in1++
      }
        while (in2 < end2) {
            write it1 to outf
            it1 = getItem(inf1); in1++
        }
        in1 += runLength; in2 += runLength;
    }
}
```

Heap Sort

We can create a heap, and repeatedly insert and remove from a heap, which is $O(2n \log n) = O(n \log n)$

Comparison between Mergesort and Quicksort

Why quicksort is better than mergesort?
There are certain reasons due to which quicksort is better especially in case of arrays:

1. **Auxiliary Space**: Mergesort uses extra space, quicksort requires little space and exhibits good cache locality.(in-place sorting)
2. **Worst Cases**: The worst case of quicksort $O(n^2)$ can be avoided by using randomized quicksort. It can be easily avoided with high probability by choosing the right pivot. Obtaining an average case behavior by choosing right pivot element makes it improvise the performance and becoming as efficient as Merge sort.
3. **Locality of reference** : Quicksort in particular exhibits good cache locality and this makes it faster

than merge sort in many cases like in virtual memory environment.

4. **Merge sort is better for large data structures**: Mergesort is a stable sort (unlike quicksort and heapsort) and can be easily adapted to operate on linked lists and very large lists stored on slow-to-access media.

Summary of Comparison-Based Sorting

- Selection sort:
  - stability depends on implementation
  - not adaptive
- Bubble sort:
  - stable if items don't move past same-key items
  - adaptive if it terminates when no swaps
- Insertion sort:
  - stability depends on implementation of insertion
  - adaptive if it stops scan when position is found
- Quicksort:
  - easy to make stable on lists; difficult on arrays
  - can be adaptive depending on implementation
- Merge sort:
  - is stable if merge oeration is stable
  - can be made adaptive (but above version is not)

Sorting Lower Bound for Comparison-Based Sorting

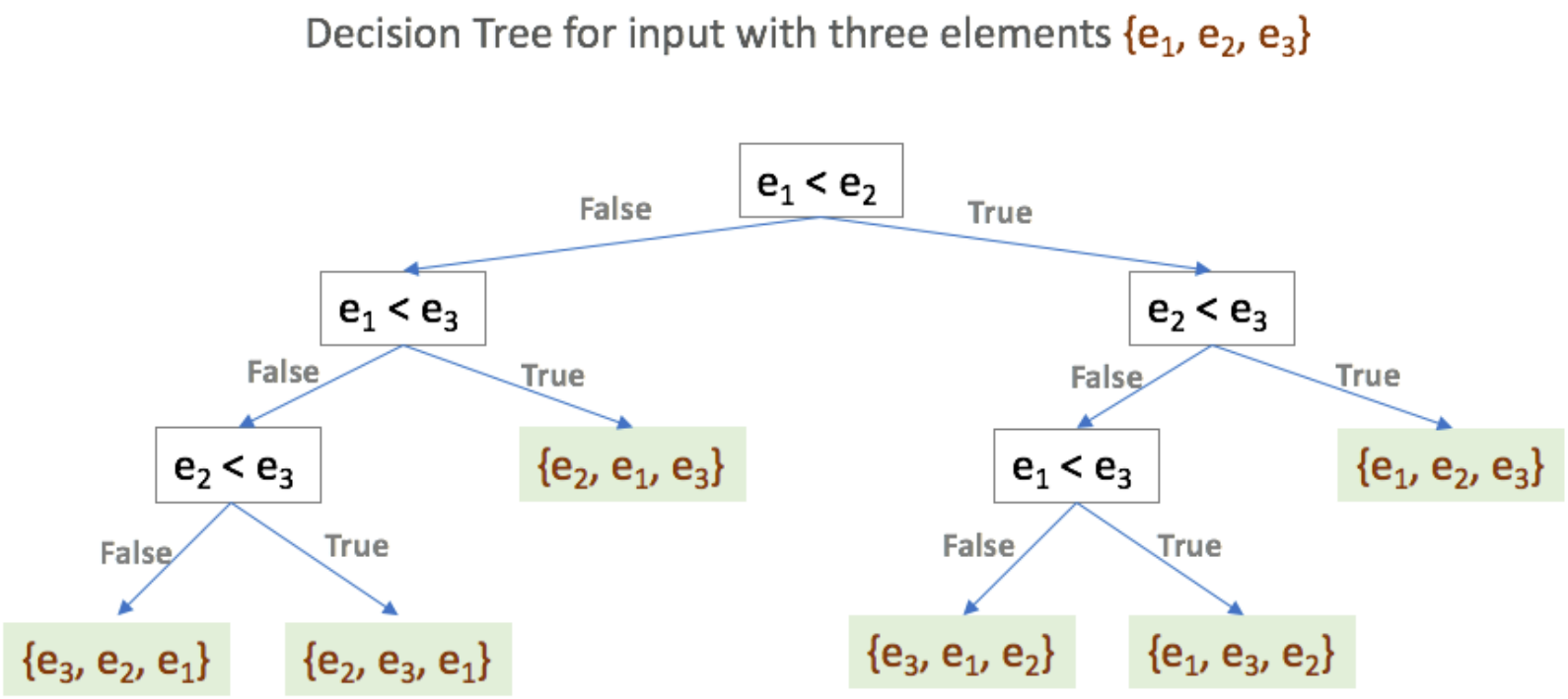
Many popular sorting algorithms "compare" pairs of keys (objects) to sort an input sequence.

Lower Bound: Any comparison-based sorting algorithm must take Ω (n log n) time to sort n elements in

the worst case.

Given n elements (no duplicates),

- there are $n!$ possible permutation sequences
- one of these possible sequences is a sorted sequence
- each comparision reduces number of possible sequences to be considered



For a given input,

- the algorithm follows a path from the root to a leaf
- requires one comparison at each level
- there are $n!$ leaves for n elements
- height of such tree is at least $\log_2(n!)$, so number of comparisions required is at least $log_2(n!)$

$$\log_2(n!) = \log_2(1) + \log_2(2) + \ldots + \log_2(n/2) + \ldots + \log_2(n-1) + \log_2(n)$$
$$\log_2(n!) >= \log_2(n/2) + \ldots + \log_2(n-1) + \log_2(n)$$
$$\log_2(n!) >= (n/2)\log_2(n/2)$$
$$\log_2(n!) = \Omega(n \log_2 n)$$

Therefore, for any comparison-based sorting algorithm, the lower bound is $\Omega(n \log_2 n)$.

Non comparison-based Sorting Algo

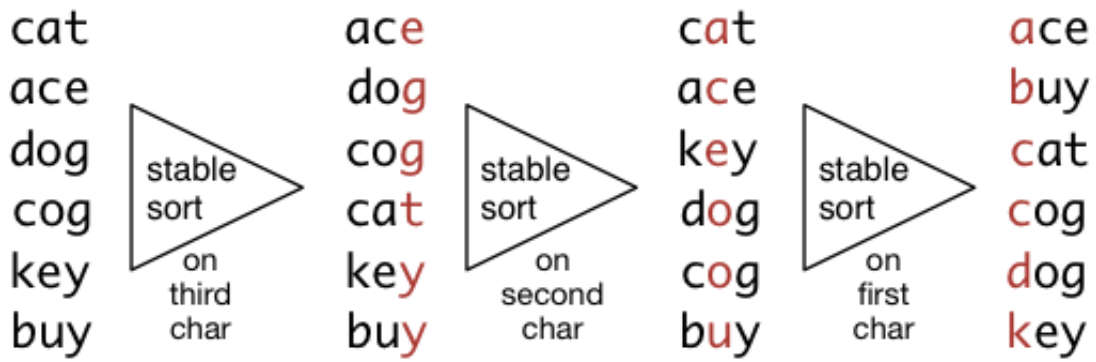
Radix Sort

Radix sort is a non-comparative sorting algorithm.

Radix sort basic idea:

- represent key as a tuple $(k_1, k_2, \ldots, k_m)$
- finite and normally few possible values of k_i

- sorting algorithm:
    - **stable** sort on k_m,
    - then **stable** sort on $k_{(m-1)}$,
    - similarly continue until k_1



Time complexity:

- stable sort like bucket/pigeonhole sort runs in time $O(n)$
- radix sort runs in time $O(mn)$, where m is number of sub-keys (k_i in a tuple above)
- radix sort performs better (for sufficiently large n) than the best comparison-based sorting algorithms

Bucket/Pigeonhole Sort

Bucket sort is a common way for the stable sorting in the radix sort.

Bucket/Pigeonhole sort basic idea:

- finite and normally few possible values of keys, for example
  - numeric: 0 to 9
  - week days: monday, tuesday, wednesday, thursday, friday, saturday, sunday
  - months: january to december
- each key value maps to an index into the array of buckets/pigeonholes
- one bucket (pigeonhole) per key value
- sorting algorithm:
  - phase-1: move each entry from the input sequence to the corresponding bucket (say queue) in

the array of buckets
    - phase-2: move entries of each bucket, in the required order, to the end of the output sequence

Time complexity: bucket sort runs in time $O(n)$, assuming number of buckets is not large.