Sorting for Algorithm Design

option	applications

sorting algorithm	• when all elements to sort are known at one time
	• many tasks can reduce to sorting, or can be done more efficiently if the elements are sorted – "when in doubt, sort"
	  • e.g. searching, closest pair, element uniqueness, finding the mode, selection
	  • e.g. greedy algorithms such as event scheduling and Kruskal's algorithm

PriorityQueue	• for sorting in a dynamic environment, where the elements are not all known at once • e.g. greedy algorithms such as Dijkstra's algorithm and Prim's algorithm

Pragmatics –

- can handle increasing vs decreasing order, keys vs records, and non-numerical data by abstracting a *comparator* from the sorting algorithm

Sorting

Key algorithms –

comparison-based

- selection sort – $O(n^2)$
  - heapsort – $O(n \log n)$
- insertion sort – $O(n^2)$
  - shellsort can be as good as $O(n \log^2 n)$ with the right gap sequence https://en.wikipedia.org/wiki/Shellsort
- quicksort, merge sort – $O(n \log n)$

distribution

- bucket sort – average $O(n)$ when $b \approx n$, $O(n+n^2/b+b)$ otherwise worst $O(n^2)$ [or $O(n \log n)$ depending on algorithm used for buckets]
  - *b* is the number of buckets ADM 4.7 https://en.wikipedia.org/wiki/Bucket_sort
- radix sort – $O(nd)$ https://en.wikipedia.org/wiki/Radix_sort
  - *d* is the number of "digits"

- external sorting https://en.wikipedia.org/wiki/External_sorting
  - when not all of the data fits in memory at once

Sorting – Key Algorithms

adapt for different sort orders by translating "less than" to "comparator says comes first"

selection sort	repeatedly pick the smallest remaining element in the unsorted portion of the array and swap it with the current position
heapsort	use a priority queue to repeatedly pick the smallest remaining element

SelectionSort(A)
  For $i = 1$ to n do
    $Sort[i]$ = Find-Minimum from A
    Delete-Minimum from A
  Return($Sort$)

heapsort is a variation of selection sort that selects from a heap instead of an unsorted array

```c
void selection_sort(item_type s[], int n) {
    int i, j;    /* counters */
    int min;     /* index of minimum */

    for (i = 0; i < n; i++) {
        min = i;
        for (j = i + 1; j < n; j++) {
            if (s[j] < s[min]) {
                min = j;
            }
        }
        swap(&s[i], &s[min]);
    }
}
```

in-place sort – `s[0..i-1]` is the sorted part of the array, `s[i..n]` is unsorted
final swap takes advantage of `s[i..n]` unsorted to remove an element without shifting

```c
void heapsort_(item_type s[], int n) {
    int i;              /* counters */
    priority_queue q;   /* heap for heapsort */

    make_heap(&q, s, n);

    for (i = 0; i < n; i++) {
        s[i] = extract_min(&q);
    }
}
```

can be implemented in-place – use a max heap instead
`s[0..n-i-1]` is the heap, `s[n-i..n]` sorted
extract max swaps the root with the last element in the array, which is exactly where it should go

Sorting – Key Algorithms

adapt for different sort orders by translating "less than" to "comparator says comes first"

insertion sort	repeatedly insert the next element into the correct position in the sorted part of the array
shellsort	repeatedly sort elements at different gap intervals, reducing the gap size over iterations

shellsort is a variation of insertion sort that sorts all elements *gap* apart in each pass

```c
# Sort an array a[0...n-1].
gaps = [701, 301, 132, 57, 23, 10, 4, 1]  # Ciura gap sequence

# Start with the largest gap and work down to a gap of 1
# similar to insertion sort but instead of 1, gap is being used in each step
foreach (gap in gaps)
{
    # Do a gapped insertion sort for every element in gaps
    # Each loop leaves a[0..gap-1] in gapped order
    for (i = gap; i < n; i += 1)
    {
        # save a[i] in temp and make a hole at position i
        temp = a[i]
        # shift earlier gap-sorted elements up until the correct location for a[i] is found
        for (j = i; (j >= gap) && (a[j - gap] > temp); j -= gap)
        {
            a[j] = a[j - gap]
        }
        # put temp (the original a[i]) in its correct location
        a[j] = temp
    }
}
```

the performance depends on the gap sequence, and can be complicated to analyze
not as efficient as quicksort, but is significantly better than $O(n^2)$ and avoids recursion

insertion sort

```c
for (i = 1; i < n; i++) {
    j = i;
    while ((j > 0) && (s[j] < s[j - 1])) {
        swap(&s[j], &s[j - 1]);
        j = j-1;
    }
}
```

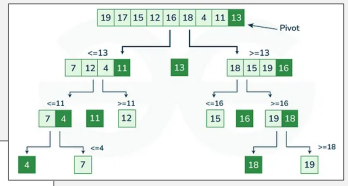
Sorting – Key Algorithms

adapt for different sort orders by translating "less than" to "comparator says comes first"

quicksort	use a pivot element to partition the array, then recursively sort the two subarrays

```
void quicksort(item_type s[], int l, int h) {
    int p;    /* index of partition */

    if (l < h) {
        p = partition(s, l, h);
        quicksort(s, l, p - 1);
        quicksort(s, p + 1, h);
    }
}
```

```
int partition(item_type s[], int l, int h) {
    int i;          /* counter */
    int p;          /* pivot element index */
    int firsthigh;  /* divider position for pivot element */

    p = h;          /* select last element as pivot */
    firsthigh = l;
    for (i = l; i < h; i++) {
        if (s[i] < s[p]) {
            swap(&s[i], &s[firsthigh]);
            firsthigh++;
        }
    }
    swap(&s[p], &s[firsthigh]);

    return(firsthigh);
}
```



leads to worst-case O(n²) performance for sorted, reverse sorted

instead pick random element as pivot for O(n log n) expected performance (low probability of worst case)

can also avoid worst case from sorted and nearly sorted arrays by shuffling first – shuffling is O(n)

https://medium.com/@dinmukhamed.dukumbayev26/quicksort-is-c-a8cc34a6ccc0

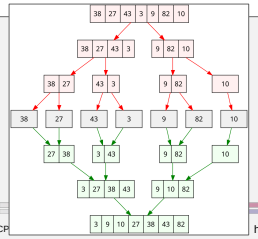
Sorting – Key Algorithms

adapt for different sort orders by translating "less than" to "comparator says comes first"

merge sort	divide the array in half, recursively sort the two halves, and then combine them into a single sorted array

```
void merge_sort(item_type s[], int low, int high) {
    int middle;    /* index of middle element */

    if (low < high) {
        middle = (low + high) / 2;
        merge_sort(s, low, middle);
        merge_sort(s, middle + 1, high);

        merge(s, low, middle, high);
    }
}
```

```
void merge(item_type s[], int low, int middle, int high) {
int i;                  /* counter */
queue buffer1, buffer2; /* buffers to hold elements for merging */

init_queue(&buffer1);
init_queue(&buffer2);

for (i = low; i <= middle; i++) enqueue(&buffer1, s[i]);
for (i = middle + 1; i <= high; i++) enqueue(&buffer2, s[i]);

i = low;
while (!(empty_queue(&buffer1) || empty_queue(&buffer2))) {
    if (headq(&buffer1) <= headq(&buffer2)) {
        s[i++] = dequeue(&buffer1);
    } else {
        s[i++] = dequeue(&buffer2);
    }
}

while (!empty_queue(&buffer1)) {
    s[i++] = dequeue(&buffer1);
}

while (!empty_queue(&buffer2)) {
    s[i++] = dequeue(&buffer2);
}
}
```



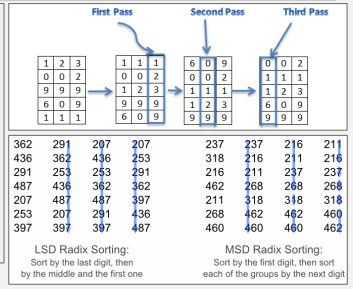
O(n log n) best/worst

multiway mergesort is suitable for external sorting – sort chunks of the data, then use 2- or k-way merging in a series of passes

https://en.wikipedia.org/wiki/Merge_sort#/media/File:Merge_sort_algorithm_diagram.svg

Sorting – Key Algorithms

bucket sort	partition the elements into bins, then sort each bin
radix sort	process elements character-by-character or digit-by-digit, typically from right to left, sorting according to the current character/digit in each pass



buckets are based on a range of values for numbers, prefixes for strings

insertion sort can be used to sort buckets as each element is added, or another sorting algorithm to sort bucket contents after distribution step

each pass of radix sort sorts by distributing into buckets based on current digit/character

LSD processes entire collection in each pass – distribution must preserve order within buckets (stable)

MSD most suitable for strings or fixed-length integers

https://codeiiest-dev.github.io/Algorithms/SortingAlgorithms/BucketSort.html
https://lifeindatastructures.wordpress.com/2017/10/19/radix-sort/