COMP2521 24T2
Sorting Algorithms (III)
Divide-and-Conquer Sorting Algorithms

Sim Mautner

cs2521@cse.unsw.edu.au

Slides adapted from those by Kevin Luxa 2521 24T1

Divide-and-Conquer Algorithms

divide-and-conquer algorithms
split a problem into two or more subproblems,
solve the subproblems recursively,
and then combine the results.

Merge Sort

Merge Sort

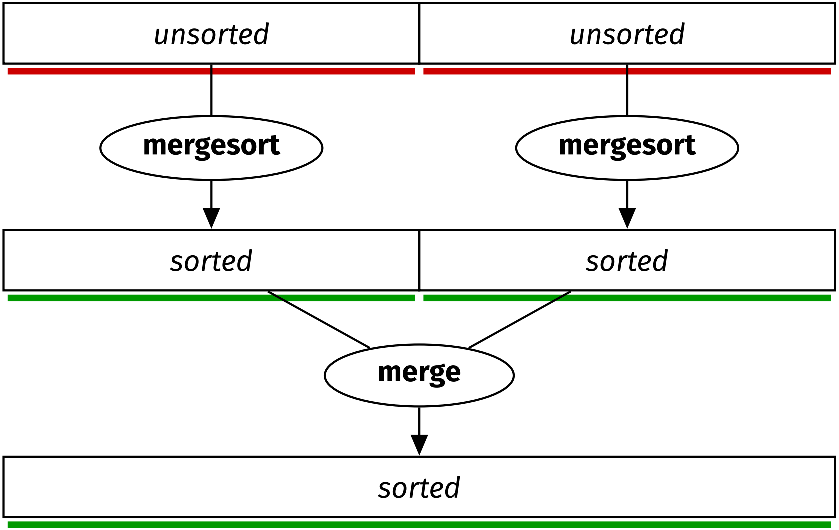
Invented by John von Neumann
in 1945

Merge Sort

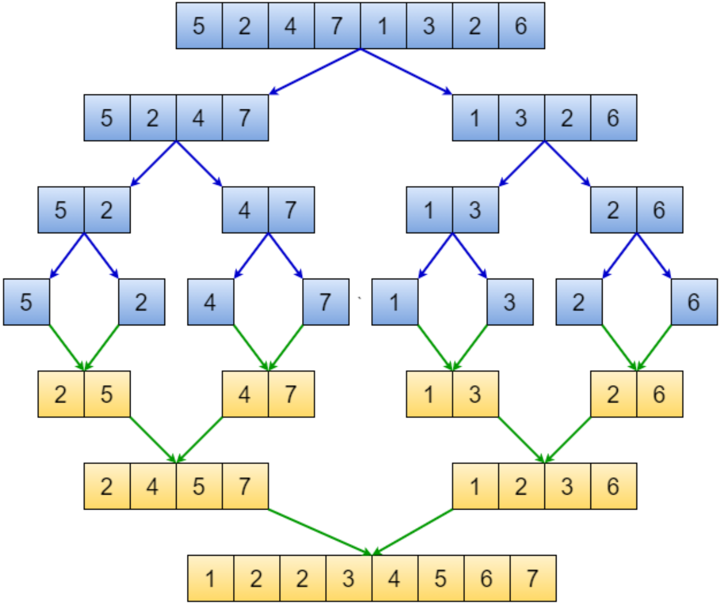
A divide-and-conquer sorting algorithm:

split the array into two roughly equal-sized parts
recursively sort each of the partitions
merge the two now-sorted partitions into a sorted array

Merge Sort


unsorted
unsorted
mergesort
mergesort
sorted
sorted
merge
sorted

Merge Sort

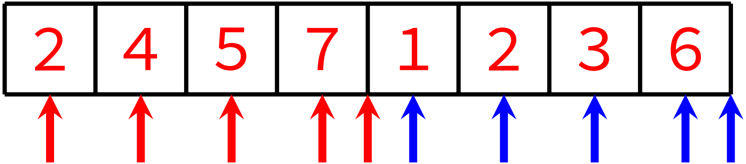

5 2 4 7 1 3 2 6
5 2 4 7
1 3 2 6
5 2
4 7
1 3
2 6
5
2
4
7
1
3
2
6
2 5
4 7
1 3
2 6
2 4 5 7
1 2 3 6
1 2 2 3 4 5 6 7

How do we split the array?

- We don't physically split the array
- We simply calculate the midpoint of the array
  - `mid = (lo + hi) / 2`
- Then recursively sort each half by passing in appropriate indices
  - Sort between indices `lo` and `mid`
  - Sort between indices `mid + 1` and `hi`
- This means the time complexity of splitting the array is $O(1)$

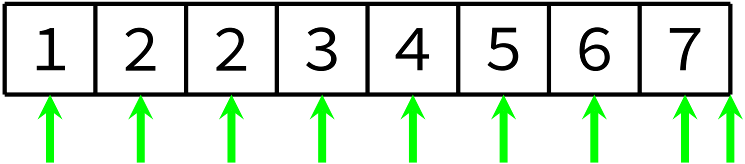
How do we merge two sorted subarrays?

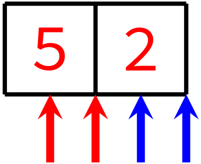
- We merge the subarrays into a *temporary array*
- Keep track of the smallest element that has not been merged in each subarray
- Copy the smaller of the two elements into the temporary array
  - If the elements are equal, take from the left subarray
- Repeat until all elements have been merged
- Then copy from the temporary array back to the original array



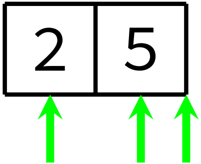
When items are equal, merge takes from the left subarray
(this ensures stability)

Now copy back to original array





Now copy back to original array



- The time complexity of merging two sorted subarrays is $O(n)$, where n is the total number of elements in both subarrays
- Therefore:
  - Merging two subarrays of size 1 takes 2 “steps”
  - Merging two subarrays of size 2 takes 4 “steps”
  - Merging two subarrays of size 4 takes 8 “steps”
  - ...

Merge Sort

C Implementation: Sort

```
void mergeSort(Item items[], int lo, int hi) {
    if (lo >= hi) return;
    int mid = (lo + hi) / 2;
    mergeSort(items, lo, mid);
    mergeSort(items, mid + 1, hi);
    merge(items, lo, mid, hi);
}
```

Merge Sort

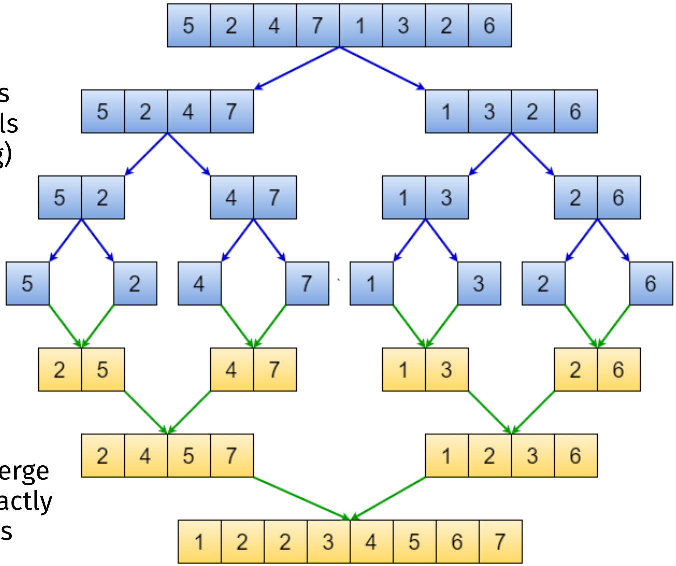
C Implementation: Merge

```
void merge(Item items[], int lo, int mid, int hi) {
    Item *tmp = malloc((hi - lo + 1) * sizeof(Item));
    int i = lo, j = mid + 1, k = 0;

    // Scan both segments, copying to `tmp'.
    while (i <= mid && j <= hi) {
        if (le(items[i], items[j])) {
            tmp[k++] = items[i++];
        } else {
            tmp[k++] = items[j++];
        }
    }

    // Copy items from unfinished segment.
    while (i <= mid) tmp[k++] = items[i++];
    while (j <= hi) tmp[k++] = items[j++];

    // Copy `tmp' back to main array.
    for (i = lo, k = 0; i <= hi; i++, k++) {
        items[i] = tmp[k];
    }

    free(tmp);
}
```

Merge Sort
Analysis

Split

$n - 1$ splits
($\log_2 n$ levels of splitting)



$O(n)$

Merge

We have to merge n numbers exactly $\log_2 n$ times

$O(n \log n)$

Analysis:

- Merge sort splits the array into equal-sized partitions halving at each level $\Rightarrow \log_2 n$ levels
- The same operations happen at every recursive level
- Each 'level' requires $\leq n$ comparisons

Therefore:

- The time complexity of merge sort is $O(n \log n)$
  - Best-case, average-case, and worst-case time complexities are all the same

Note: Not required knowledge in COMP2521!

Let $T(n)$ be the time taken to sort n elements.

Splitting arrays into two halves takes constant time.
Merging two sorted arrays takes n steps.

So we have that:
$T(n) = 2T(n/2) + n$

Then the Master Theorem (see COMP3121) can be used to show that the time complexity is $O(n \log n)$.

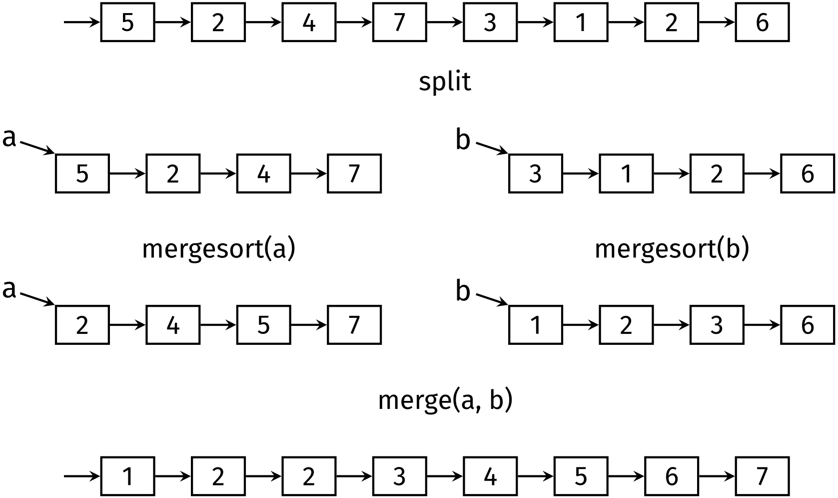
Stable
Due to taking from left subarray if items are equal during merge

Non-adaptive
$O(n \log n)$ best case, average case, worst case

Not in-place
Merge uses a temporary array of size up to n
Note: Merge sort also uses $O(\log n)$ stack space

Merge Sort on Lists

It is possible to apply merge sort on linked lists.



Bottom-Up Merge Sort

An approach that works non-recursively!

- On each pass, our array contains sorted *runs* of length m.
- Initially, n sorted runs of length 1.
- The first pass merges adjacent elements into runs of length 2.
- The second pass merges adjacent elements into runs of length 4.
- Continue until we have a single sorted run of length n.

Can be used for *external* sorting;
e.g., sorting disk-file contents

Bottom-Up Merge Sort

Example

	[0]	[1]	[2]						.....							[15]
Original	A	S	O	R	T	I	N	G	E	X	E	M	P	L	A	R
After 1st pass *sorted slices of length 2*	A	S	O	R	I	T	G	N	E	X	E	M	L	P	A	R
After 2nd pass *sorted slices of length 4*	A	O	R	S	G	I	N	T	E	E	M	X	A	L	P	R
After 3rd pass *sorted slices of length 8*	A	G	I	N	O	R	S	T	A	E	E	L	M	P	R	X
After 4th pass *sorted slice of length 16*	A	A	E	E	G	I	L	M	N	O	P	R	R	S	T	X

Bottom-Up Merge Sort

C Implementation


```c
void mergeSortBottomUp(Item items[], int lo, int hi) {
    for (int m = 1; m <= hi - lo; m *= 2) {
        for (int i = lo; i <= hi - m; i += 2 * m) {
            int end = min(i + 2 * m - 1, hi);
            merge(items, i, i + m - 1, end);
        }
    }
}
```

Quick Sort

Quick Sort

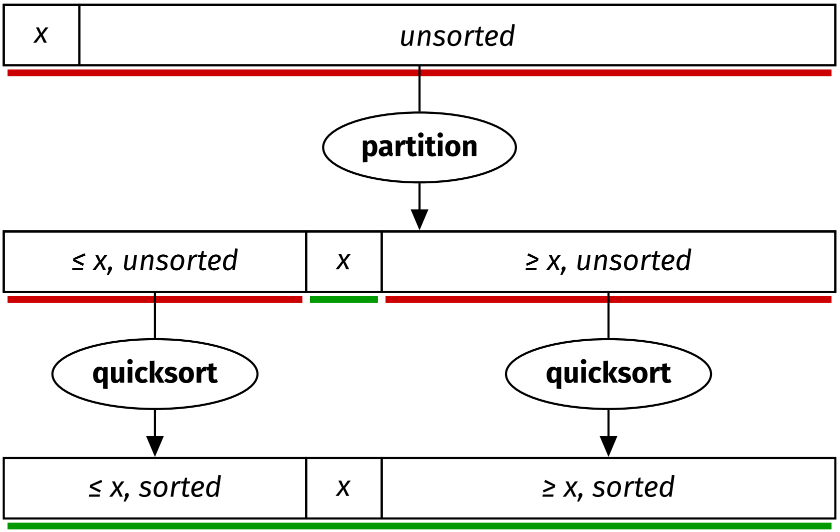
Invented by Tony Hoare
in 1959

Quick Sort

Method:

1. Choose an item to be a pivot
2. Rearrange (partition) the array so that
    - All elements to the left of the pivot are less than (or equal to) the pivot
    - All elements to the right of the pivot are greater than (or equal to) the pivot
3. Recursively sort each of the partitions

Quick Sort

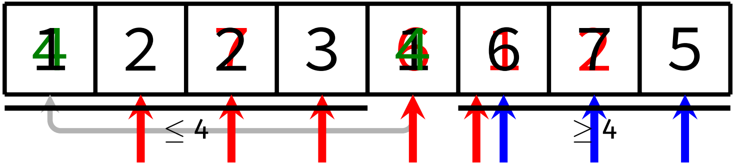

x
unsorted
partition
≤ x, unsorted
x
≥ x, unsorted
quicksort
quicksort
≤ x, sorted
x
≥ x, sorted

Partitioning

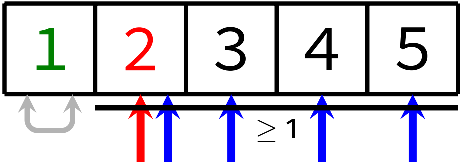
How to partition an array?

- Assume the pivot is stored at index `lo`
- Create index `l` to start of array (`lo + 1`)
- Create index `r` to end of array (`hi`)
- Until `l` and `r` meet:
  - Increment `l` until `a[l]` is greater than pivot
  - Decrement `r` until `a[r]` is less than pivot
  - Swap items at indices `l` and `r`
- Swap the pivot with index `l` or `l - 1` (depending on the item at index `l`)

Until the indices meet:



Until the indices meet:



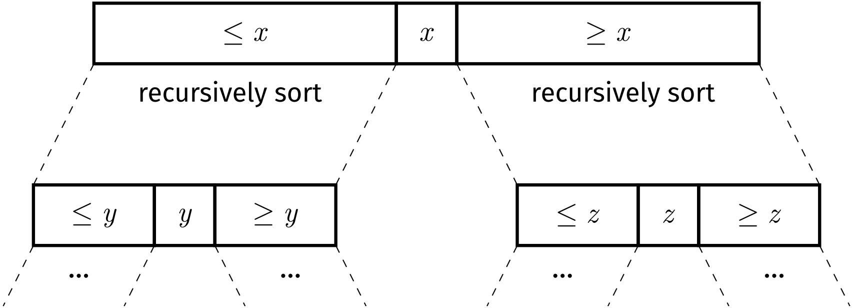
- Partitioning is $O(n)$, where n is the number of elements being partitioned
  - About n comparisons are performed, at most $\frac{n}{2}$ swaps are performed

```c
void naiveQuickSort(Item items[], int lo, int hi) {
    if (lo >= hi) return;
    int pivotIndex = partition(items, lo, hi);
    naiveQuickSort(items, lo, pivotIndex - 1);
    naiveQuickSort(items, pivotIndex + 1, hi);
}
```

```c
int partition(Item items[], int lo, int hi) {
    Item pivot = items[lo];

    int l = lo + 1;
    int r = hi;
    while (l < r) {
        while (l < r && le(items[l], pivot)) l++;
        while (l < r && ge(items[r], pivot)) r--;
        if (l == r) break;
        swap(items, l, r);
    }

    if (lt(pivot, items[l])) l--;
    swap(items, lo, l);
    return l;
}
```

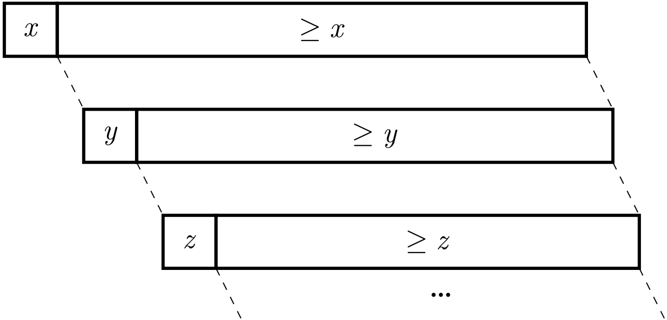
Best case: $O(n \log n)$

- Choice of pivot gives two equal-sized partitions
- Same happens at every recursive call
  - Resulting in $\log_2 n$ recursive levels
- Each "level" requires approximately n comparisons



Worst case: $O(n^2)$

- Always choose lowest/highest value for pivot
  - Resulting in partitions of size 0 and $n-1$
  - Resulting in n recursive levels
- Each "level" requires one less comparison than the level above



Average case: $O(n \log n)$

- If array is randomly ordered, chance of repeatedly choosing a bad pivot is very low
- Can also show empirically by generating random sequences and sorting them

Unstable
Due to long-range swaps

Non-adaptive
$O(n \log n)$ average case, sorted input does not improve this

In-place
Partitioning is done in-place
Stack depth is $O(n)$ worst-case, $O(\log n)$ average

Issues with Quick Sort

Choice of pivot can have a significant effect:

- Ideal pivot is the median value
- Always choosing largest/smallest $\Rightarrow$ worst case

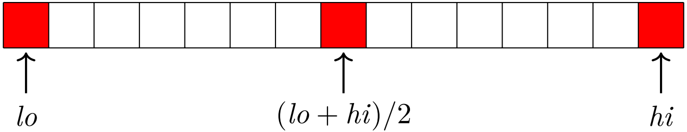
Therefore, always picking the first or last element as pivot is not a good idea:

- Existing order is a worst case
- Existing reverse order is a worst case
- Will result in partitions of size $n-1$ and 0
- This pivot selection strategy is called naïve quick sort

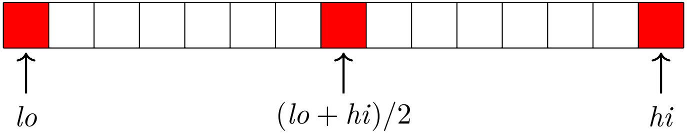
Quick Sort with Median-of-Three Partitioning

Pick three values: left-most, middle, right-most.
Pick the median of these three values as our pivot.

Ordered data is no longer a worst-case scenario.
In general, doesn't eliminate the worst-case ...
... but makes it much less likely.



Quick Sort with Median-of-Three Partitioning



1. Sort a$[lo]$, a$[(lo + hi)/2]$, a$[hi]$, such that a$[(lo + hi)/2] \leq$ a$[lo] \leq$ a$[hi]$
2. Partition on a$[lo]$ to a$[hi]$

Quick Sort with Median-of-Three Partitioning

Example

Which element is selected as the pivot?

Answer: 5

5	3	7	2	1	4	6	8

lo $(lo + hi)/2$ hi

Quick Sort with Median-of-Three Partitioning

C Implementation


```
void medianOfThreeQuickSort(Item items[], int lo, int hi) {
    if (lo >= hi) return;
    medianOfThree(items, lo, hi);
    int pivotIndex = partition(items, lo, hi);
    medianOfThreeQuickSort(items, lo, pivotIndex - 1);
    medianOfThreeQuickSort(items, pivotIndex + 1, hi);
}

void medianOfThree(Item a[], int lo, int hi) {
    int mid = (lo + hi) / 2;
    if (gt(a[mid], a[lo])) swap(a, mid, lo);
    if (gt(a[lo],  a[hi])) swap(a, lo,  hi);
    if (gt(a[mid], a[lo])) swap(a, mid, lo);
    // now, we have a[mid] <= a[lo] <= a[hi]
}
```

Quick Sort with Randomised Partitioning

Idea: Pick a random value for the pivot

This makes it *nearly* impossible to systematically generate inputs that would lead to $O(n^2)$ performance

Quick Sort with Randomised Partitioning

C Implementation


```c
void randomisedQuickSort(Item items[], int lo, int hi) {
    if (lo >= hi) return;
    swap(items, lo, randint(lo, hi));
    int pivotIndex = partition(items, lo, hi);
    randomisedQuickSort(items, lo, pivotIndex - 1);
    randomisedQuickSort(items, pivotIndex + 1, hi);
}

int randint(int lo, int hi) {
    int i = rand() % (hi - lo + 1);
    return lo + i;
}
```

Note: `rand()` is a pseudo-random number generator provided by `<stdlib.h>`.
The generator should be initialised with `srand()`.

Insertion Sort Improvement

For small sequences (when $n < 5$, say),
quick sort is expensive
because of the recursion overhead.

Solution: Handle small partitions with insertion sort

Insertion Sort Improvement

C Implementation - Version 1


```c
#define THRESHOLD 5

void quickSort(Item items[], int lo, int hi) {
    if (hi - lo < THRESHOLD) {
        insertionSort(items, lo, hi);
        return;
    }

    medianOfThree(items, lo, hi);
    int pivotIndex = partition(items, lo, hi);
    quickSort(items, lo, pivotIndex - 1);
    quickSort(items, pivotIndex + 1, hi);
}
```

Insertion Sort Improvement

C Implementation - Version 2


```c
#define THRESHOLD 5

void quickSort(Item items[], int lo, int hi) {
    doQuickSort(items, lo, hi);
    insertionSort(items, lo, hi);
}

void doQuickSort(Item items[], int lo, int hi) {
    if (hi - lo < THRESHOLD) return;

    medianOfThree(items, lo, hi);
    int pivotIndex = partition(items, lo, hi);
    doQuickSort(items, lo, pivotIndex - 1);
    doQuickSort(items, pivotIndex + 1, hi);
}
```

Quick Sort on Lists

It is possible to quick sort a linked list:

1. Pick first element as pivot
   - Note that this means ordered data is a worst case again
   - Instead, can use median-of-three or random pivot
2. Create two empty linked lists A and B
3. For each element in original list (excluding pivot):
   - If element is less than (or equal to) pivot, add it to A
   - If element is greater than pivot, add it to B
4. Recursively sort A and B
5. Form sorted linked list using sorted A, the pivot, and then sorted B

Quick Sort vs Merge Sort

Design of modern CPUs mean,
for sorting arrays in RAM
quick sort *generally* outperforms merge sort.

Quick sort is more ‘cache friendly’:
good locality of access on arrays.

On the other hand, merge sort is
readily stable, readily parallel,
a good choice for sorting linked lists

Summary of Divide-and-Conquer Sorts

	Time complexity			Properties	
	Best	Average	Worst	Stable	Adaptive
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Yes	No
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	No	No

Feedback

https://forms.office.com/r/riGKCze1cQ