COMP2521 23T3
Sorting Algorithms (III)

Kevin Luxa
cs2521@cse.unsw.edu.au

quick sort

Merge sort uses a trivial split operation;
all the heavy lifting is in the *merge* operation.

Can we split the collection in a more intelligent way,
so combining the results is easier?

...*e.g.*, making sure all elements in one part
are less than elements in the second part?

Quick sort!

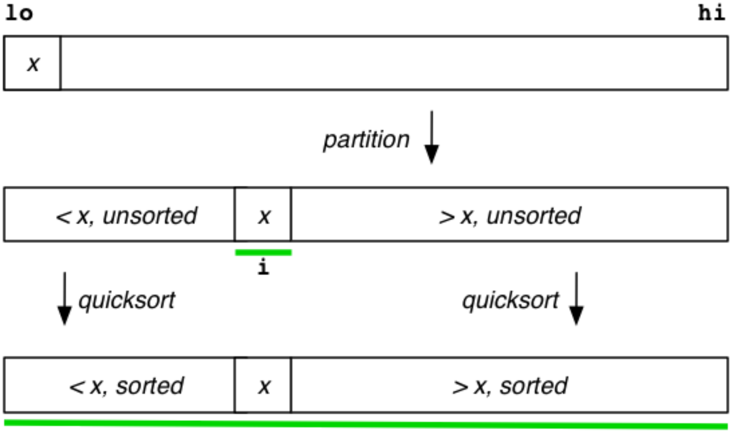
Invented by Tony Hoare

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


lo
hi
x
partition
< x, unsorted
x
> x, unsorted
i
quicksort
quicksort
< x, sorted
x
> x, sorted

Partitioning

How do we 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`)

Partitioning

Example

4	2	7	3	6	1	2	5

Partitioning

Example

Increment left index while element is $\leq$ pivot

4	2	7	3	6	1	2	5

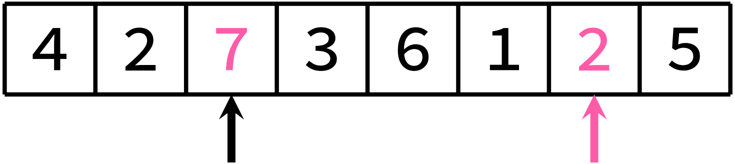
Increment left index while element is $\leq$ pivot

4	2	7	3	6	1	2	5

Decrement right index while element is $\geq$ pivot

4	2	7	3	6	1	2	5

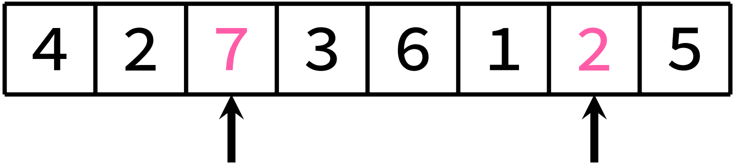
Decrement right index while element is $\geq$ pivot



Partitioning

Example

Swap the two elements

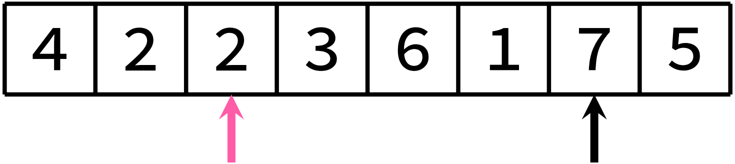


Swap the two elements

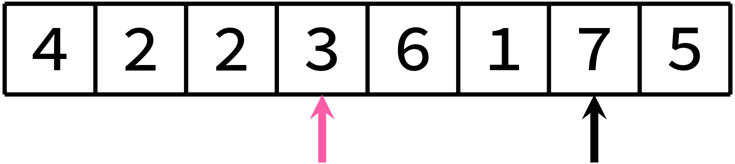
4	2	2	3	6	1	7	5

Partitioning
Example

Repeat until the indices meet



Repeat until the indices meet



Partitioning
Example

Repeat until the indices meet

4	2	2	3	6	1	7	5

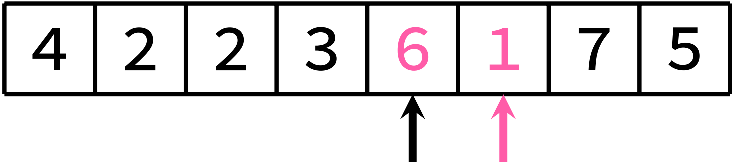
Partitioning

Example

Repeat until the indices meet

4	2	2	3	6	1	7	5

Repeat until the indices meet



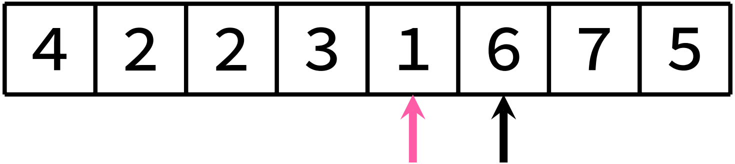
Partitioning

Example

Repeat until the indices meet

4	2	2	3	1	6	7	5

Repeat until the indices meet



Repeat until the indices meet

4	2	2	3	1	6	7	5

Swap the pivot into the middle (be careful!)

4	2	2	3	1	6	7	5

Swap the pivot into the middle (be careful!)

1	2	2	3	4	6	7	5

Partitioning
Example

Done

1	2	2	3	4	6	7	5

≤ 4 ≥ 4

- 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);
}
```

Quick Sort

C Implementation: Partition

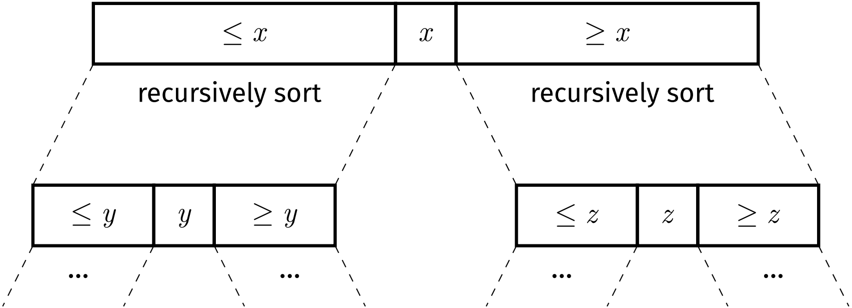
```c
int partition(Item items[], int lo, int hi) {
    Item pivot = items[lo];

    int l = lo + 1;
    int r = hi;
    while (true) {
        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;
}
```

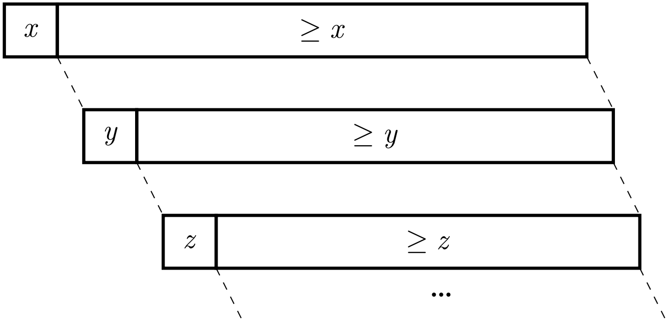
Best case: $O(n \log n)$

- 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

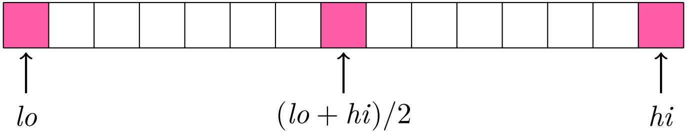
Therefore, always picking the first or last element as pivot is not a good idea:

- 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

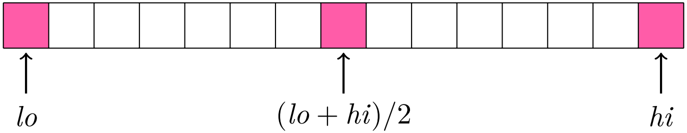
Quick Sort with Median-of-Three Partitioning

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$[lo] \leq$ a$[(lo+hi)/2] \leq$ a$[hi]$
2. Swap a$[lo]$ and a$[(lo+hi)/2]$
3. Partition on a$[lo]$ to a$[hi]$

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

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


```
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


```
#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

```
#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

Feedback

https://forms.office.com/r/aPF09YHZ3X