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

Method

Partitioning
Implementation
Analysis
Properties
Issues
Median-of-Three
Partitioning
Randomised
Partitioning
Improvements
Sorting Lists

Diagram:

- **Partitioning**:
  - Median-of-Three Partitioning
  - Randomised Partitioning

- **Issues**:
  - Improve

- **Sorting Lists**:
  - Quick Sort

- **Diagram**:
  - Initial list: `<x, unsorted`
  - Partition:
    - `<x, unsorted`
    - `x`
    - `>x, unsorted`
  - Quicksort:
    - `<x, sorted`
    - `x`
    - `>x, sorted`
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 $l$)
Median-of-Three Partitioning

Randomised Partitioning

Improve-ments

Sorting Lists
Increment left index while element is $\leq$ pivot
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

4 2 7 3 6 1 2 5
Swap the two elements

4 2 7 3 6 1 2 5

Swap the two elements

4 2 7 3 6 1 2 5
Swap the two elements

4 2 2 3 6 1 7 5
Repeat until the indices meet

4 2 2 3 6 1 7 5
Repeat until the indices meet

4 2 2 3 6 1 7 5

Median-of-Three Partitioning
Randomised Partitioning
Improvements
Sorting Lists
Partitioning Example

Repeat until the indices meet

4 2 2 3 6 1 7 5
Repeat until the indices meet
Repeat until the indices meet

\[
\begin{array}{cccccccc}
4 & 2 & 2 & 3 & 6 & 1 & 7 & 5 \\
\end{array}
\]
Repeat until the indices meet
Repeat until the indices meet

4 2 2 3 1 6 7 5
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

Median-of-Three Partitioning
Randomised Partitioning
Improvements

Sorting Lists

Method
Partitioning
Example
Analysis
Implementation
Analysis
Properties
Issues
Median-of-Three Partitioning
Randomised Partitioning
Improvements
Sorting Lists

Done

\[ \begin{array}{ccccccc}
1 & 2 & 2 & 3 & 4 & 6 & 7 & 5 \\
\hline
\leq 4 & & & & & \geq 4
\end{array} \]
• 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
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);
}
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

![Diagram of quick sort recursion](attachment://quick_sort_diagram.png)
**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
Method
Partitioning
Implementation
Analysis
Properties
Issues
Median-of-Three Partitioning
Randomised Partitioning
Improvements
Sorting Lists

Quick Sort
Properties

| Method          | Partitioning
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Properties</td>
<td></td>
</tr>
<tr>
<td>Issues</td>
<td></td>
</tr>
<tr>
<td>Median-of-Three Partitioning</td>
<td></td>
</tr>
<tr>
<td>Randomised Partitioning</td>
<td></td>
</tr>
<tr>
<td>Improvements</td>
<td></td>
</tr>
<tr>
<td>Sorting Lists</td>
<td></td>
</tr>
</tbody>
</table>

**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
Choice of pivot can have a significant effect:

- Ideal pivot is the median value
- Always choosing largest/smallest ⇒ 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.

$$lo \quad (lo + hi)/2 \quad hi$$
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

```c
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

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

Solution: Handle small partitions with insertion sort.
#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);
}
#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);
}

void insertionSort(Item items[], int lo, int hi) {
}
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.
https://forms.office.com/r/aPF09YHZ3X