Cosc 241
Programming and Problem Solving
Lecture 17 (29/4/2019)
Quicksort

Lech Szymanski
lechszym@cs.otago.ac.nz

Keywords: sorting, quicksort

UNIVERSITY of OTAGO
SAPERE AUDE
Te Whare Wānanga o Otāgo
NEW ZEALAND

The limits of sorting performance

- Algorithms which sort based only on the results of comparisons between their data are called comparison sorts.
- Since each comparison can have only three different outcomes (less than, equal, greater than) a set of k comparisons can have at most 3^k outcomes (in fact if the data are all distinct, this decreases to 2^k).
- If we have n distinct data items they might be in any one of $n!$ orders. Any two different orders require different rearrangements of the data to sort them.
- So, unless $n! \leq 2^k$ we cannot hope to sort every possible ordering of n distinct data items with k comparisons.
- This leads to the conclusion that a constant multiple of $n \log n$ comparisons are *necessary* to guarantee the ability to sort n items.
- Are they sufficient?

Sorting performance

- Both selection sort and insertion sort require a constant multiple of n^2 comparisons in the worst case, so they don't achieve the theoretical lower bound.
- We will consider three more sorting algorithms: quicksort, mergesort and (eventually) heapsort.
- The latter two provide $O(n \log n)$ worst case performance.
- The first does not - it degrades to $O(n^2)$ performance for some orderings of the data items. However, it has both historical and practical significance.
- Practically if we know the data is likely to be "random" then the average case performance is $O(n \log n)$.
- (A minor variation of) quicksort forms the core of the Java system sort - so it's useful to know about.

Quicksort

- Quicksort is ancient in computing terms - it was developed in 1959 (published 1961) by Tony Hoare.
- The fundamental idea is very simple:
  - Choose one of your data items called the pivot.
  - Split the remaining data into the items smaller than the pivot and the items larger than (or equal to) the pivot.
  - Put all the small items before the pivot and all the large items after the pivot.
  - Sort those two groups individually (using quicksort of course).
- But, there are some nice clever tricks for doing this in an array without using any significant amount of extra storage.

Pseudocode for quicksort

We want to sort a subarray of an array A from positions lo to hi. We make use of an auxiliary algorithm called partition which modifies the subarray and returns an index p such that:

- all elements of the (modified) subarray before p are less than $A[p]$, and
- all elements after p are greater than or equal to $A[p].$

quicksort(A, lo, hi):

if $hi - lo \leq 1$ **then**
  return (*the subarray is empty or a singleton*)
end if
$p \leftarrow$ partition(A, lo, hi)
quicksort(A, lo, p) (*exclusive of p*)
quicksort($A, p + 1, hi$)

Building quicksort

- If you look at the wikipedia page on Quicksort you will see that there are several common partition schemes as well as alternative schemes for selecting the pivot:

  > *The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance.*

- But that is engineering not theory! We'll stick with a simple choice in both instances that works well enough.
- Note too that some choices require slight changes to the pseudocode though the underlying idea is the same.

Our choices

- The pivot will be the first element of the subarray being sorted (i.e., the element in position *lo*).
- We remove this element and store its value - this creates a “hole” at position *lo*.
- Starting at the other end ($hi - 1$) we scan from right to left until we find an element smaller than the pivot. We exchange that element and the “hole”. Now the elements to the right of the hole belong above the pivot.
- The element we just swapped belongs to the left of the pivot. So we start just to the right of it scanning to the right and looking for elements at least as big as the pivot. When we find one, we exchange it with the “hole” and return to right to left scanning.
- When either boundary reaches the hole, we put the pivot element in at the final position of the hole and return that value.

Partitioning code

```
private int partition(int[] a, int lo, int hi) {
  int pivot = a[lo];
  int hole = lo;
  int left = lo+1;
  int right = hi-1;
  while (true){
    while (right > hole && a[right] >= pivot) right--;
    if (right == hole) break;
    a[hole] = a[right];
    hole = right;
    while (left < hole && a[left] < pivot) left++;
    if (left == hole) break;
    a[hole] = a[left];
    hole = left;
  }
  a[hole] = pivot;
  return hole;
}
```

Sorting code

```
public void sort(int[] a) {
  quicksort(a, 0, a.length);
};

private void quicksort(int[] a, int lo, int hi) {
 if (hi - lo <= 1) return;
 int p = partition(a, lo, hi);
 quicksort(a, lo, p);
 quicksort(a, p+1, hi);
}
```

Complexity and remarks

- Formally the best we can say for `quicksort` is that it's $O(n^2)$ because if the original data is already in sorted order we:
  - First check that everything is correctly placed relative to the pivot
  - Recursively sort two subarrays of size 0 and $n-1$ respectively.
- The first part is $O(n)$ so we wind up with something like $n + (n-1) + (n-2) + \cdots$ which is $O(n^2)$.
- On random data though `quicksort` generally behaves much much better than that in fact with $O(n \log n)$ behaviour.
- We'll explore this more thoroughly next time in the context of discussing divide and conquer algorithms.