Cosc 241
Programming and Problem Solving
Lecture 19 (6/5/2019)
Divide and conquer algorithms
Mergesort

Lech Szymanski

lechszym@cs.otago.ac.nz

Keywords: divide and conquer, mergesort

UNIVERSITY of OTAGO

Te Whare Wānanga o Otāgo

NEW ZEALAND

What is ‘divide and conquer’?

- A divide and conquer algorithm working on a problem of size parameter n works as follows:
  - (Pre) Break the problem apart into two or more smaller problems whose size parameters add up to at most n,
  - (Rec) Solve those problems recursively,
  - (Post) Combine those solutions into a solution of the original problem.
- E.g., for `quicksort`
  - (Pre) Select the pivot and partition the array “before the pivot” and “after the pivot” (total size $n - 1$),
  - (Rec) Sort the parts before and after the pivot,
  - (Post) Not required.

Mergesort

- `Mergesort` is another divide and conquer algorithm for sorting arrays.
  - (Pre) Split the array into two pieces of nearly equal size,
  - (Rec) Sort the pieces,
  - (Post) Merge the results together.
- To merge two sorted arrays is easy: look at the smallest element of each. Take the smaller one and put it in the result, now look at the smallest remaining elements ...
- `Mergesort` is the preferred method for sorting large stacks of exam papers (dropping back to insertion sort when the stack size is manageable).

When does divide and conquer work well?

- Short answer - when all the subproblems are of size at most cn for some constant $c < 1$, **and** the total time needed in (Pre) and (Post) is $O(n)$.
- In that case we get $O(n \log n)$ performance.
- Strictly speaking that's a bit more than what's needed but it's the practical version.
- "The subproblems should be a constant fraction smaller than the main problem and the work required to create and combine them should be linear".
- `Mergesort` clearly meets this requirement (basically $c = 1/2$ or a tiny bit more works).
- `Quicksort` sometimes fails - if the data is already sorted the subproblems are of size 0 and $n - 1$ (and we saw this was a problem case.)

The layer cake in the good case

- Think of the different calls we wind up making to the algorithm as being arranged in layers.

Layer 0	The main call.
Layer 1	All the calls made in (Rec) from Layer 0.
Layer 2	All the calls made in (Rec) from Layer 1.
…	…

- The total size of all the calls in a single layer is at most n. Therefore the total amount of (Pre) and (Post) work done in a single layer is $O(n)$.
- The total number of layers is $O(\log n)$ since the maximum possible size of a problem in Layer k is $c^k n$ and $c < 1$. When $k > -\log n / \log c$ this is less than 1 and there is no Layer $k + 1$.
- Total work done is (work per layer) times (number of layers) and is $O(n \log n)$.

Mergesort sort code

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

private void sort(int[] a, int left, int right) {
  if (right - left <= 1) return;
  int mid = (left + right)/2;
  sort(a, left, mid);
  sort(a, mid, right);
  merge(a, left, mid, right);
}
```

Mergesort merge code

```
private void merge(int[] a, int left,
                              int mid, int right) {
  int[] temp = new int[right-left];
  int leftPos = left; int rightPos = mid; int i = 0;
  while (leftPos < mid && rightPos < right) {
    if (a[leftPos] < a[rightPos]) {
      temp[i++] = a[leftPos++];
    } else {
      temp[i++] = a[rightPos++];
    }
  }
  while (leftPos < mid) {
    temp[i++] = a[leftPos++];
  }
  while (rightPos < right) {
    temp[i++] = a[rightPos++];
  }
  System.arraycopy(temp, 0, a, left, right-left);
}
```

Room for improvement

- There are several places where we could tinker with `Mergesort` to speed it up in practice.
- The most obvious is delegating to insertion sort when `right - left` is less than some pre-set threshold.
- Also we could maintain a single static array `temp` and reuse it in the various calls to `merge` – this saves some time in object creation and garbage collection.
- We'll explore some of these ideas and compare our various sorting methods in Lecture 22 .