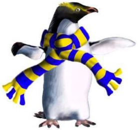
Cosc 201
Algorithms and Data Structures
Lecture 7 (17/3/2024)
Mergesort revisited

Brendan McCane
brendan.mccane@otago.ac.nz
and
Michael Albert



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.

Historical and technical digression

- Quicksort was designed in 1959 - a time when sorting in place was important.
- It still has significant use when sorting elements of primitive type.
- Memory access can be localised and the comparisons are direct.
- However, those advantages are limited when sorting objects of reference type.
- In that case each element of the array is just a reference to where the object *really* is.
- So there are no local access advantages.
- Average case for quicksort is $n \log n$, but worst case is n^2. See board demo.
- Can we do as well or better with some other algorithm?

Can we do better?

- Quicksort's worst case is when the pivot is the smallest or largest element.
- In that case, the partitioning is as unbalanced as possible.
- Can we ensure that any partitioning is as balanced as possible?

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.

This algorithm is trivial except for the merging step.

Merging

Suppose that you were given two arrays already in sorted order and asked to produce a third one containing all the elements of the two given arrays and also in sorted order. How would you do that?

2	3	6	8	9

4	6	7

2	3	4	6	6	7	8	9

So how do we merge?

Given: *a* and *b* are sorted arrays, *m* is an array whose size is the sum of their sizes.

Desired Outcome: The elements of *a* and *b* have been copied into *m* in sorted order.

- Maintain indices, *ai*, *bi* and *mi* of the 'active' location in *a*, *b* and *m*.
- If both *ai* and *bi* represent actual indices of *a* and *b*, find the one which points to the lesser value (break ties in favour of *a*) copy that value into *m* at *mi* and increment *mi* and whichever of *ai* or *bi* was used for the copy.
- Once one of *ai* and *bi* is out of range, copy the rest of the other array into the remainder of *m*.

Merge in Java

```
public static int[] merge(int[] a, int[] b) {

   int[] m = new int[a.length + b.length];
   int ai = 0, bi = 0, mi = 0;

   while (ai < a.length && bi < b.length) {
     if (a[ai] <= b[bi]) m[mi++] = a[ai++];
     else                m[mi++] = b[bi++];
   }

   while (ai < a.length) m[mi++] = a[ai++];
   while (bi < b.length) m[mi++] = b[bi++];

   return m;
}
```

Time complexity of merge

- The size parameter n (total size of the input) is equal to the sum of the lengths of a and b.
- There's no obvious counter-controlled loop - indeed there are three separate while statements, with different sorts of control.
- The key observation is that each time we go once through a loop (any of the three) the parameter *mi* increases by one, and this parameter runs from 0 to $n - 1$.
- Since the total number of loop bodies executed is (always exactly) n and each involves a constant amount of work, the total time complexity is $\Theta(n)$.

Mergesort

Recall the definition of Mergesort:

(Pre) Split the array into two pieces of nearly equal size,

(Rec) Sort the pieces,

(Post) Merge the results together.

- How can we analyse its time complexity?
- There are no counters!
- We know the time complexity of the Pre and Post phases (Pre is constant, Post is $\Theta(n)$ where n is the sum of the sizes of the pieces being merged)
- So

$$M(n) = \Theta(n) + 2 \times M(n/2)$$

Ignoring all the problems

I'm going to pretend that $\Theta(n)$ is literally $C \times n$ for some constant C and chase the recurrence a bit farther.

$$
\begin{aligned}
M(n) &= \boxed{C \times n + 2 \times M(n/2)} \\
&= C \times n + 2 \times (C \times (n/2) + 2 \times M(n/4)) \\
&= \boxed{C \times (2n) + 4 \times M(n/4)} \\
&= C \times (2n) + 4 \times (C \times (n/4) + 2 \times M(n/8)) \\
&= \boxed{C \times (3n) + 8 \times M(n/8)} \\
&= \ldots \quad \text{and after } k \text{ applications} \\
&= \boxed{C \times (kn) + 2^k \times M(n/2^k)} \, .
\end{aligned}
$$

But where does it all end?

Where does it end?

- It ends when we hit a base case of the algorithm.
- In the simplest version this is just when we get to $n/2^k = 1$.
- More generally, we might split out earlier than this, but regardless, the work done in a base case is (bounded by) some constant, D.
- So, if k is large enough that $n/2^k$ is a base case, we get:

$$M(n) = C \times (kn) + 2^k \times D$$

- But how big is k?
- Certainly $2^k \leqslant n$ so $k \leqslant log(n)$ which gives

$$M(n) \leqslant C \times (n\,log(n)) + Dn = \Theta(n\,log(n)).$$

The master theorem

- There's a Master theorem that allows one to analyse the complexity of almost all divide and conquer situations that arise in practice.
- It's a bit complicated, but for our purposes:

Theorem

In a divide and conquer algorithm where the pre- and post-processing work are $O(n)$ and the division is into parts of size at least cn for some constant $0 < c < 1$ the total time-complexity is $O(n \log n)$ and generally $\Theta(n \log n)$.

Theorem

In a divide and conquer algorithm where the pre- and post-processing work are $O(n)$ and the division is into parts of size c on one side, and $n - c$ on the other side, for some constant $1 \leq c < n$, the total time-complexity is $O(n^2)$.