COMP90038 Algorithms and Complexity

Sorting with Divide-and-Conquer

Michael Kirley

Lecture 11

Semester 1, 2016

Divide and Conquer

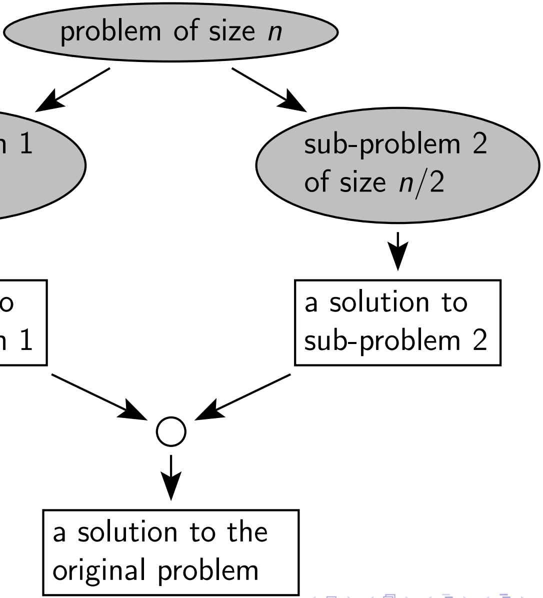
We earlier studied recursion as a powerful problem solving technique.

The divide-and-conquer strategy tries to make the most of this:

1. Divide the given problem instance into smaller instances.
2. Solve the smaller instances recursively.
3. Combine the smaller solutions to solve the original instance.

This works best when the smaller instances can be made to be of equal size.

Split-Solve-and-Join Approach


problem of size n
sub-problem 1 of size $n/2$
sub-problem 2 of size $n/2$
a solution to sub-problem 1
a solution to sub-problem 2
a solution to the original problem

Divide-and-Conquer Algorithms

We will discuss:

- The Master Theorem
- Mergesort
- Quicksort
- Tree traversal
- Closest Pair revisited

Divide-and-Conquer Recurrences

What is the time required to solve a problem of size n by divide-and-conquer?

For the general case, assume we split the problem into b instances (each of size n/b), of which a need to be solved:

$$T(n) = aT(n/b) + f(n)$$

where $f(n)$ expresses the time spent on dividing a problem into b sub-problems and combining the a results.

(A very common case is $T(n) = 2T(n/2) + n$.)

How to find closed forms for these recurrences?

The Master Theorem

(A proof is in Levitin's Appendix B.)

For integer constants $a \geq 1$ and $b > 1$, and function f with $f(n) \in \Theta(n^d)$, $d \geq 0$, the recurrence

$$T(n) = aT(n/b) + f(n)$$

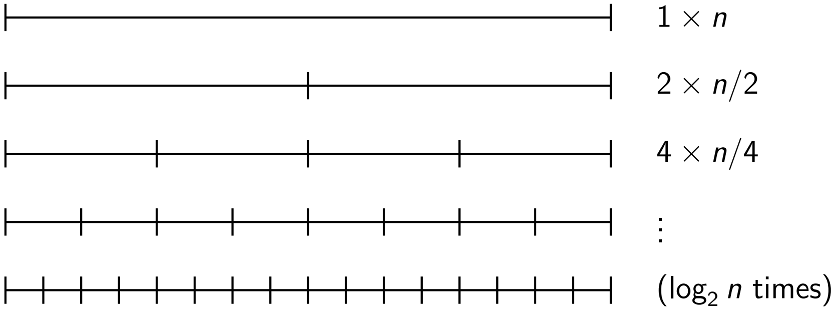
(with $T(1) = c$) has solutions, and

$$T(n) = \begin{cases} \Theta(n^d) & \text{if } a < b^d \\ \Theta(n^d \log n) & \text{if } a = b^d \\ \Theta(n^{\log_b a}) & \text{if } a > b^d \end{cases}$$

Note that we also allow a to be greater than b.

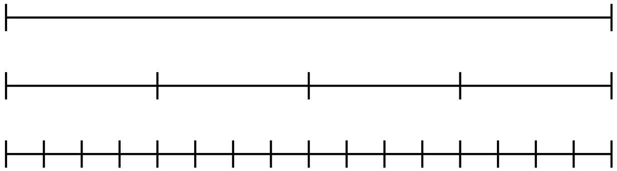
Master Theorem: Example 1

$$T(n) = 2T(n/2) + n \qquad a = 2, b = 2, d = 1$$



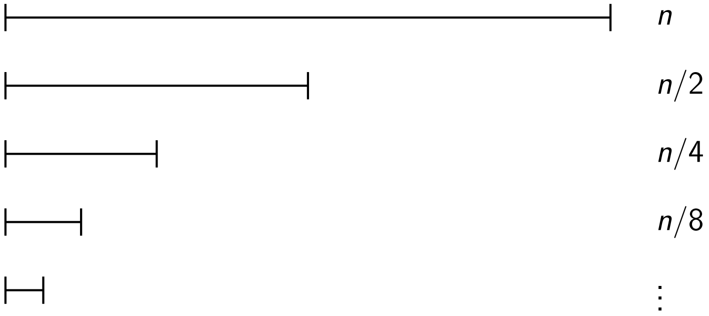
Master Theorem: Example 2

$$T(n) = 4T(n/4) + n \qquad a = 4, b = 4, d = 1$$



Master Theorem: Example 3

$T(n) = T(n/2) + n \qquad a = 1, b = 2, d = 1$

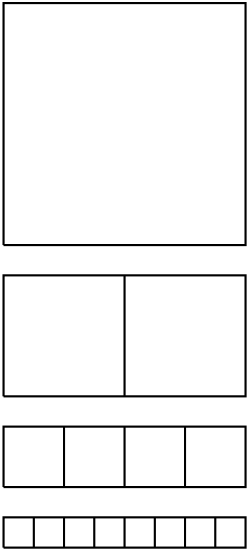


Master Theorem: Example 4

$$T(n) = 2T(n/2) + n^2$$

$$a = 2, b = 2, d = 2$$

Here $a < b^d$ and we simply get n^d.



Mergesort

Perhaps the most obvious application of divide-and-conquer:

To sort an array (or a list), cut it into two halves, sort each half, and merge the two results.

```
function $\textsc{Mergesort}(A[0..n-1])$
    if $n > 1$ then
        copy $A[0..\lfloor n/2 \rfloor - 1]$ to $B[0..\lfloor n/2 \rfloor - 1]$
        copy $A[\lfloor n/2 \rfloor..n-1]$ to $C[0..\lceil n/2 \rceil - 1]$
        $\textsc{Mergesort}(B[0..\lfloor n/2 \rfloor - 1])$
        $\textsc{Mergesort}(C[0..\lceil n/2 \rceil - 1])$
        $\textsc{Merge}(B, C, A)$
```

Mergesort: Merging Arrays

```
**function** MERGE($B[0..p-1], C[0..q-1], A[0..p+q-1]$)
    $i \leftarrow 0; j \leftarrow 0; k \leftarrow 0$
    **while** $i < p$ and $j < q$ **do**
        **if** $B[i] \leq C[j]$ **then**
            $A[k] \leftarrow B[i]$
            $i \leftarrow i + 1$
        **else**
            $A[k] \leftarrow C[j]$
            $j \leftarrow j + 1$
        $k \leftarrow k + 1$
    **if** $i = p$ **then**
        copy $C[j..q-1]$ to $A[k..p+q-1]$
    **else**
        copy $B[i..p-1]$ to $A[k..p+q-1]$
```

Mergesort Analysis

How many comparisons will $\textsc{Merge}$ need to make in the worst case, when given arrays of size $\lfloor n/2 \rfloor$ and $\lceil n/2 \rceil$?

If the largest and second-largest elements are in different arrays, then $n - 1$ comparisons. Hence the cost equation for $\textsc{Mergesort}$ is

$$C(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2C(n/2) + n - 1 & \text{otherwise} \end{cases}$$

By the Master Theorem, $C(n) \in \Theta(n \log n)$.

Mergesort Properties

For large n, the number of comparisons made tends to be around 75% of the worst-case scenario.

Is mergesort stable?

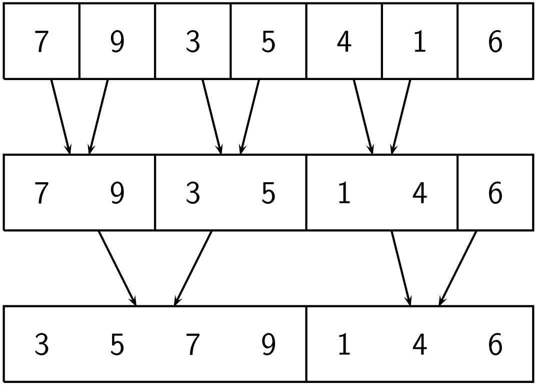
Is mergesort in-place?

If comparisons are fast, mergesort ranks between quicksort and (next week's) heapsort for time, assuming random data.

Mergesort is the method of choice for linked lists and for very large collections of data.

Bottom-Up Mergesort

An alternative way of doing mergesort:
Generate runs of length 2, then of length 4, and so on:



Quicksort

Quicksort takes a divide-and-conquer approach that is different to mergesort's.

It uses the partitioning idea from QUICKSELECT, picking a pivot element, and partitioning the array around that, so as to obtain this situation:

$$\underbrace{A[0]\ldots A[s-1]}_{\text{all are } \leq A[s]} \quad A[s] \quad \underbrace{A[s+1]\ldots A[n-1]}_{\text{all are } \geq A[s]}$$

The element $A[s]$ will be in its final position (it is the $(s+1)$th smallest element).

All that then needs to be done is to sort the segment to the left, recursively, as well as the segment to the right.

Quicksort

Very short and elegant:

```
function $\textsc{Quicksort}(A[lo..hi])$
    if $lo < hi$ then
        $s \leftarrow \textsc{Partition}(A[lo..hi])$
        $\textsc{Quicksort}(A[lo..s-1])$
        $\textsc{Quicksort}(A[s+1..hi])$
```

Hoare Partitioning

This is the standard way of doing partitioning for quicksort:

```
function PARTITION($A[lo..hi]$)
    $p \leftarrow A[lo]$; $i \leftarrow lo$; $j \leftarrow hi$
    repeat
        while $i < hi$ and $A[i] \leq p$ do $i \leftarrow i + 1$
        while $j \geq lo$ and $A[j] > p$ do $j \leftarrow j - 1$
        $swap(A[i], A[j])$
    until $i \geq j$
    $swap(A[i], A[j])$ — undo the last swap
    $swap(A[lo], A[j])$ — bring pivot to its correct position
    return $j$
```

Quicksort Analysis—Best Case Analysis

The best case happens when the pivot is the median; that results in two sub-tasks of equal size.

$$C_{best}(n) = \begin{cases} 0 & \text{if } n < 2 \\ 2C_{best}(n/2) + n & \text{otherwise} \end{cases}$$

The 'n' is for the n key comparisons performed by PARTITION.

By the Master Theorem, $C(n) \in \Theta(n \log n)$, just as for mergesort, so quicksort's best case is (asymptotically) no better than mergesort's worst case.

Quicksort Analysis—Worst Case Analysis

The worst case happens if the array is already sorted.

In that case, we don't really have divide-and-conquer, because each recursive call deals with a problem size that has only been decremented by 1:

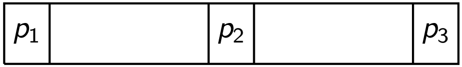
$$C_{worst}(n) = \begin{cases} 0 & \text{if } n < 2 \\ C_{worst}(n-1) + n & \text{otherwise} \end{cases}$$

That is, $C_{worst}(n) = n + (n-1) + \cdots + 3 + 2 \in \Theta(n^2)$.

Quicksort Improvements: Median-of-Three

It would be better if the pivot was chosen randomly.

A cheap and useful approximation to this is to take the median of three candidates, $A[lo]$, $A[hi]$, and $A[\lfloor(lo+hi)/2\rfloor]$.



Reorganise the three elements so that p_1 is the median, and p_3 is the largest of the three.

Now run quicksort as before.

Quicksort Improvements: Median-of-Three

In fact, with median-of-three, we can have a much faster version than before, simplifying tests in the innermost loops:

```
function PARTITION($A[lo..hi]$)
    $p \leftarrow A[lo]$; $i \leftarrow lo$; $j \leftarrow hi + 1$
    repeat
        ~~while $i < hi$ and $A[i] \leq p$ do $i \leftarrow i + 1$~~
        repeat $i \leftarrow i + 1$ until $A[i] \geq p$
        ~~while $j \geq lo$ and $A[j] > p$ do $j \leftarrow j - 1$~~
        repeat $j \leftarrow j - 1$ until $A[j] \leq p$
        $swap(A[i], A[j])$
    until $i \geq j$
    $swap(A[i], A[j])$
    $swap(A[lo], A[j])$
    return $j$
```

Quicksort Improvements: Early Cut-Off

A second useful improvement is to stop quicksort early and switch to insertion sort. This is easily implemented:

```
function SORT($A[lo..hi]$)
    QUICKALMOSTSORT($A[lo..hi]$)
    INSERTIONSORT($A[lo..hi]$)

function QUICKALMOSTSORT($A[lo..hi]$)
    if $lo + 10 < hi$ then
        $s \leftarrow$ PARTITION($A[lo..hi]$)
        QUICKALMOSTSORT($A[lo..s-1]$)
        QUICKALMOSTSORT($A[s+1..hi]$)
```

Quicksort Properties

With these (and other) improvements, quicksort is considered the best available sorting method for arrays of random data.

A major reason for its speed is the very tight inner loop in PARTITION.

Although mergesort has a better performance guarantee, quicksort is faster on average.

In the best case, we get $\lceil \log_2 n \rceil$ recursive levels. It can be shown that on random data, the expected number is $2 \log_e n \approx 1.38 \log_2 n$. So quicksort's average behaviour is very close to the best-case behaviour.

Is quicksort stable?

Is it in-place?

Next Up

Tree traversal methods, plus we apply the divide-and-conquer technique to the closest-pair problem.