THE UNIVERSITY OF MELBOURNE

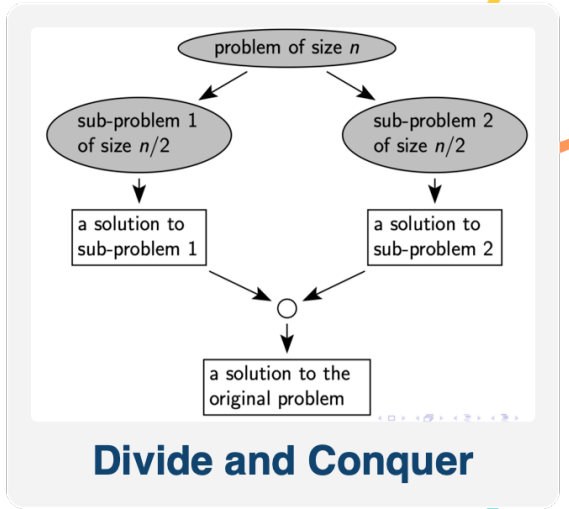
COMP90038 Algorithms and Complexity

Tutorial 6 Divide and Conquer

Tutor: Chuang(Frank) Wang

THE UNIVERSITY OF MELBOURNE

1. Review



Divide and Conquer

Master Theorem

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.

Mergesort

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.

```
procedure MERGESORT(A[·], n)                          ▷ Sort A[0]..A[n − 1]
    if n > 1 then
        for i ← 0 to ⌊n/2⌋ − 1 do                     ▷ Copy left half of A to B
            B[i] ← A[i]
        for i ← 0 to ⌈n/2⌉ − 1 do                     ▷ Copy right half of A to C
            C[i] ← A[⌊n/2⌋ + i]
        MERGESORT(B, ⌊n/2⌋)                           ▷ Sort B
        MERGESORT(C, ⌈n/2⌉)                           ▷ Sort C
        MERGE(B, ⌊n/2⌋, C, ⌈n/2⌉, A)                 ▷ Merge B and C into A
```

Mergesort Analysis

How many comparisons will 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 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? **Yes!**

Is mergesort in-place? **No!**

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

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

Quicksort

Quicksort

Very short and elegant:

```
procedure QUICKSORT(A[·], lo, hi)
    if lo < hi then
        s ← PARTITION(A, lo, hi)
        QUICKSORT(A, lo, s − 1)
        QUICKSORT(A, s + 1, hi)
```

Initial call: QUICKSORT$(A, 0, n - 1)$.

Hoare Partitioning

This is the standard way of doing partitioning for quicksort:

```
function PARTITION(A[·], lo, hi)
    p ← A[lo]; i ← lo; j ← hi
    repeat
        while i < hi and A[i] ≤ p do i ← i + 1
        while j ≥ lo and A[j] > p do j ← j − 1
        swap(A[i], A[j])
    until i ≥ j
    swap(A[i], A[j])                                  ▷ Undo the last swap
    swap(A[lo], A[j])                                 ▷ Bring pivot to its correct position
    return j
```

Best Case

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, $C_{best}(n) \in \Theta(n \log n)$, just as for mergesort, so quicksort's best case is (asymptotically) no better than mergesort's worst case.

Worst Case

Quicksort Analysis—Worst Case Analysis

The pivot is smallest.
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:

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

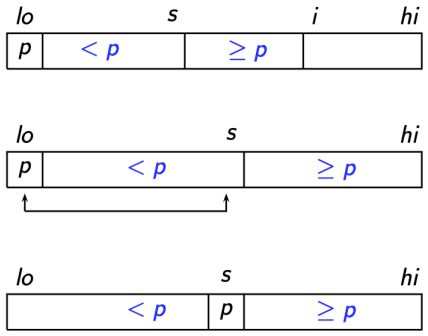
That is, $C_{worst}(n) = n + (n-1) + \cdots + 3 + 2 \in \Theta(n^2)$.

2. Tutorial Questions

Question 42

Lomuto Partitioning

```
function LomutoPartition(A[·], lo, hi)
    p ← A[lo]
    s ← lo
    for i ← lo + 1 to hi do
        if A[i] < p then
            s ← s + 1
            swap(A[s], A[i])
    swap(A[lo], A[s])
    return s
```



Finding the *k*th Smallest Element

Here is how we can use partitioning to find the *k*th smallest element.

```
function QuickSelect(A[·], lo, hi, k)
    s ← LomutoPartition(A, lo, hi)
    if s − lo = k − 1 then
        return A[s]
    else
        if s − lo > k − 1 then
            QuickSelect(A, lo, s − 1, k)
        else
            QuickSelect(A, s + 1, hi, (k − 1) − (s − lo))
```

42. Trace how QuickSelect finds the median of 39, 23, 12, 77, 48, 61, 55.

Median is the 4th smallest element. K = 4

The algorithm will stop when placing the pivot in position 4 – 1 = 3.

39	23	12	77	48	61	55
0	1	2	3	4	5	6

12	23	39	77	48	61	55
0	1	2	3	4	5	6
lo		s				hi

s – lo < k - 1

QUICKSELECT(A, 2 + 1, hi, (k - 1) – (s - lo))

QUICKSELECT(A, 3, 6, 1)

77	48	61	55
3	4	5	6

55	48	61	77
3	4	5	6
lo			hi
			s

s – lo > k - 1

QUICKSELECT(A, lo, s-1, k)

QUICKSELECT(A, 3, 5, 1)

55	48	61
3	4	5

48	55	61
3	4	5
lo	s	hi

s – lo > k - 1

QUICKSELECT(A, lo, s-1, k)

QUICKSELECT(A, 3, 3, 1)

48
3

Question 43

43. We can use QUICKSELECT to find the smallest element of an unsorted array. How does it compare to the more obvious way of solving the problem, namely scanning through the array and maintaining a variable *min* that holds the smallest element found so far?

```
function find_smallest(A)
    result =A[0]
    for i ← 1 to len_A - 1 do
        if A[i] < result:
            result = A[i]
    return result
```

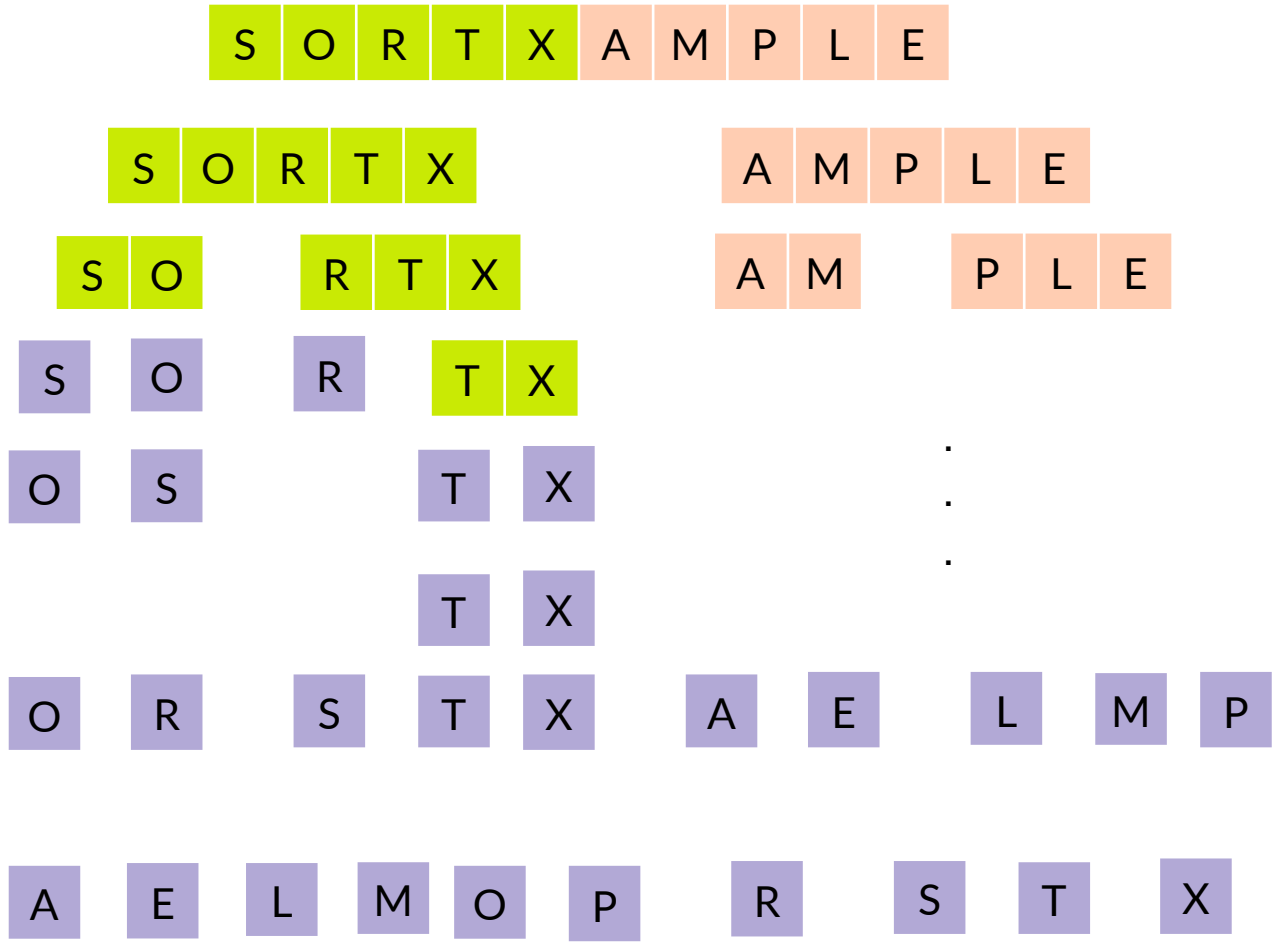
The straight-forward way of scanning through the array is better. It will require n – 1 comparisons.

QuickSelect will require that number of comparisons just to do the first round of partitioning, and of course one round may not be enough. In fact, in the worst case we end up doing $\Theta(n^2)$ comparisons.

Question 44

44. Apply mergesort to the list S, O, R, T, X, A, M, P, L, E.

- sort the left half of the array
- sort the right half of the array
- Merge the two halves together



Question 46

46. Use the Master Theorem to find the order of growth for the solutions to the following recurrences. In each case, assume $T(1) = 1$, and that the recurrence holds for all $n > 1$.

(a) $T(n) = 4T(n/2) + n$

(b) $T(n) = 4T(n/2) + n^2$

(c) $T(n) = 4T(n/2) + n^3$

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.

a = 4, b = 2, d = 1 a > b ^ d

(a) $T(n) = \Theta(n^{\log_2 4}) = \Theta(n^2)$.

a = 4, b = 2, d = 2 a = b ^ d

(b) $T(n) = \Theta(n^2 \log n)$.

a = 4, b = 2, d = 3 a < b ^ d

(c) $T(n) = \Theta(n^3)$.

Question 49

49. Let T be defined recursively as follows:

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n-1) + n/2 \quad n > 1 \end{aligned}$$

The division is exact division, so $T(n)$ is a rational, but not necessarily natural, number. For example, $T(3) = 7/2$. Use telescoping to find a closed form definition of T.

Telescoping the recursive clause:

$$\begin{aligned} T(n) &= T(n-1) + n/2 \\ &= T(n-2) + (n-1)/2 + n/2 \\ &= T(n-3) + (n-2)/2 + (n-1)/2 + n/2 \\ &= T(2) + 3/2 + \ldots + (n-2)/2 + (n-1)/2 + n/2 \\ &= T(1) + 1 + 3/2 + \ldots + (n-2)/2 + (n-1)/2 + n/2 \\ &= 1 + 1 + 3/2 + \ldots + (n-2)/2 + (n-1)/2 + n/2 \\ &= 2 + \textstyle\sum_{i=3}^{n} i/2 \\ &= 2 + (\textstyle\sum_{i=3}^{n} i)/2 \\ &= 2 + ((n+3)(n-2)/2)/2 \\ &= 2 + \frac{(n+3)(n-2)}{4} \\ &= \frac{n^2+n+2}{4} \end{aligned}$$

Question 47

47. When analysing quicksort in the lecture, we noticed that an already sorted array is a worst-case input. Is that still true if we use median-of three pivot selection?

Quicksort Analysis—Worst Case Analysis

The pivot is smallest.

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:

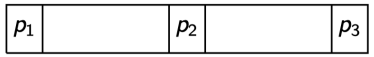
$$C_{worst}(n) = \begin{cases} 0 & \text{if } n < 2 \\ C_{worst}(n-1) + n & \text{otherwise} \end{cases}$$

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 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, $C_{best}(n) \in \Theta(n \log n)$, just as for mergesort, so quicksort's best case is (asymptotically) no better than mergesort's worst case.

This is no longer a worst case; in fact it becomes a best case!

In this case the median-of-three is in fact the array's median.

Hence each of the two recursive calls will be given an array of length at most n/2, where n is the length of the whole array.

And the arrays passed to the recursive calls are again already-sorted, so the phenomenon is invariant throughout the calls.

THE UNIVERSITY OF MELBOURNE

Next Week:

Trees, Priority queues, heaps and heapsort

Thank you !