Algorithms and Complexity

Divide and conquer

Julián Mestre

School of Information Technologies
The University of Sydney

Divide and conquer

Paradigm for designing algorithm based on the following idea:

- *Break* big problem into two or more smaller subproblems
- *Recursively* call algorithm on subproblems
- *Combine* smaller solutions into a solution for bigger problem

Correctness usually follows easily from the algorithm itself

Time complexity can be captured by recurrence, which we must solve either by hand, by using a "Master Theorem", or by using computer algebra software

Sorting

Given an array A with n numbers, sort them in increasing value

Mergesort works as follows:

- if n > 1 break A into two halves: B and C
- Sort B and C recursively
- Combine (merge) the sorted arrays B and C into A

THE UNIVERSITY OF SYDNEY

Mergesort

```
def mergesort(A):

  if |A| > 1
    B = copy of the first half of A
    C = copy of the second half of A
    mergesort(B)
    mergesort(C)
    i = j = 0
    while i < |B| or j < |C|
      if i < |B| and (j = |C| or B[i] <= C[j])
        A[i+j] = B[i]
        i += 1
      if j < |C| and (i = |B| or C[j] <= B[i])
        A[i+j] = C[j]
        j += 1
  return A
```

Time complexity

Let $T(n)$ be the running time of the algorithm, then

- $T(n) = 2T(n/2) + O(n)$ for $n > 1$
- $T(n) = O(1)$ for $n = 1$

We "unroll" the recurrence to get an asymptotic bound on $T(n)$

Unrolling the recurrence

General strategy for solving recurrences:

- Analyze the first few levels
- Identify the patter for a generic level
- Sum up over all levels

To verify the solution, we can substitute guess into the recurrence and prove it formally using induction

For Mergesort this method yields $T(n) = O(n \log n)$

There is a "Master theorem" that can handle most recurrences of interest, but unrolling will be enough for our purposes

Integer multiplication

Given two n-digit integers x and y, compute the product $x\ y$

Suppose we wanted to do it by hand. We assume that two digits can be multiplied or added in constant time (by ourselves)

In primary school we all learn an algorithm for this problem, which performs $\Theta(n^2)$ operations

While this seems like recreational mathematics, it does have real applications: Imagine multiplying two integers with 1 million digits

Recursive algorithm for IM

Let $x = x_1 2^{n/2} + x_0$ and $y = y_1 2^{n/2} + y_0$. Then

$$x\,y = x_1 y_1 2^n + x_1 y_0 2^{n/2} + x_0 y_1 2^{n/2} + x_0 y_0$$

We can compute the product of n-digit numbers by making 4 recursive calls on $n/2$-digit numbers and then combining the solutions to the subproblems.

THE UNIVERSITY OF SYDNEY

Multiplication algorithm

```
def multiply(x, y):
  // x and y are positive integers represented in binary

  if x == 0 or y == 0
    return 0
  if x == 1
    return y
  if y == 1
    return x

  // recursive case
  let $x_1$ and $x_0$ be such that $x = x_1 2^{n/2} + x_0$
  let $y_1$ and $y_0$ be such that $y = y_1 2^{n/2} + y_0$

  return multiply($x_1$, $y_1$) $2^n$ +
         (multiply($x_1$, $y_0$) + multiply($x_0$, $y_1$)) $2^{n/2}$ +
         multiply($x_0$, $y_0$)
```

Time complexity

We can compute the product of n-digit numbers by making 4 recursive calls on $n/2$-digit numbers and then spending $O(n)$ time combining the solutions to the subproblems.

$$T(n) = 4\,T(n/2) + O(n),$$

which solves to $T(n) = O(n^2)$.

No better than the simpler algorithm!

THE UNIVERSITY OF SYDNEY

Geometric series

Let r be a positive real and k a positive integer then

$$1 + r + r^2 + \dots + r^k = (r^{k+1} - 1)/(r-1)$$

Consequently if $r > 1$ then

$$1 + r + r^2 + \dots + r^k < r^{k+1}$$

and if $r < 1$ then

$$1 + r + r^2 + \dots + r^k < 1/ (1-r)$$

Another recursive algorithm

Let $x = x_1 2^{n/2} + x_0$ and $y = y_1 2^{n/2} + y_0$. Then

$$x\,y = x_1 y_1 2^n + (x_1 y_0 + x_0 y_1) 2^{n/2} + x_0 y_0$$

$$(x_1 + x_0)(y_1 + y_0) = x_1 y_1 + x_1 y_0 + x_0 y_1 + x_0 y_0$$

We can compute the product of n-digit numbers by making 3 recursive calls on $n/2$-digit numbers and then combining the solutions to the subproblems.

Another multiplication algorithm

```
def multiply(x, y):

  ⋮

  // recursive case
  let $x_1$ and $x_0$ be such that $x = x_1 2^{n/2} + x_0$
  let $y_1$ and $y_0$ be such that $y = y_1 2^{n/2} + y_0$

  first_term = multiply($x_1$, $y_1$)
  last_term = multiply($x_0$, $y_0$)
  other_term = multiply($x_1 + x_0$, $y_1 + y_0$)

  return first_term $2^n$ +
         (other_term - first_term - second_term) $2^{n/2}$ +
         second_term
```

Time complexity

We can compute the product of n-digit numbers by making 3 recursive calls on $n/2$-digit numbers and then spending $O(n)$ time combining the solutions to the subproblems.

$$T(n) = 3\,T(n/2) + O(n)$$

which solves to $T(n) = O(n^{\log_2 3})$, where $\log_2 3 \approx 1.6$

Better than the simpler algorithm!

THE UNIVERSITY OF SYDNEY

Logarithms

Base exchange rule:

$$\log_a x = (\log_b x)/(\log_b a)$$

Product rule:

$$\log_a (xy) = (\log_a x) + (\log_a y)$$

Power rule:

$$\log_a x^n = n \log_a x$$

Selection problem

Given an unsorted array A with n number and a number k, find k-th smallest number in A

Trivial solution: Sort the elements and return kth element.

Can we do better than O(n log n) ?

How could we solve this problem with divide and conquer?

First attempt

Suppose we could compute the median element of A in $O(n)$ time

- If $k < n/2$ then find k-th among elements smaller than the median
- If $k > n/2$ then find $(k-n/2)$-th among elements larger than the median

This leads to the recurrence $T(n) = T(n / 2) + O(n)$,
which solves to $T(n) = O(n)$

But how could we compute the median?

Approximating the median

We don't need the exact median. Suppose we could find in $O(n)$ time an element x in A such that

$$|A| / 3 < \text{rank}(A, x) < 2 |A| / 3$$

Then we get the recurrence

$$T(n) = T(2 n / 3) + O(n)$$

Which again solves to $T(n) = O(n)$

To approximate the median we can use a recursive call!

Median of 3-medians

Consider the following procedure

- Partition A into |A| / 3 groups of 3
- For each group find the median
- Let x be the median of the medians

We claim that x has the desired property

$$|A| / 3 < \text{rank}(A, x) < 2|A| / 3$$

Half of the groups have a median that is smaller than x, and each group has two elements smaller than x, thus

$$\#\text{ elements smaller than } x > 2\ (|A| / 6) = |A| / 3$$

$$\#\text{ elements greater than } x > 2\ (|A| / 6) = |A| / 3$$

Median of 3-medians

We don't need the exact median. With a recursive call on $n / 3$ elements, we can find x in A such that

$$|A| / 3 < \text{rank}(A, x) < 2\,|A| / 3$$

Then we get the recurrence

$$T(n) = T(2\,n / 3) + T(n / 3) + O(n)$$

Which solves to $T(n) = O(n \log n)$

No better than sorting!

Median of 5-medians

We don't need the exact median. With a recursive call on $n / 5$ elements, we can find x in A such that

$$3 |A| / 10 < \text{rank}(A, x) < 7 |A| / 10$$

Then we get the recurrence

$$T(n) = T(7 n / 10) + T(n / 5) + O(n)$$

Which solves to $T(n) = O(n)$

Asymptotically faster than sorting!