CS 341: Algorithms

Douglas R. Stinson

David R. Cheriton School of Computer Science
University of Waterloo

Winter, 2015

Table of Contents

Recurrence Relations

Suppose $a_1, a_2, \ldots,$ is an infinite sequence of real numbers.

A recurrence relation is a formula that expresses a general term a_n in terms of one or more previous terms $a_1, \ldots, a_{n-1}$.

A recurrence relation will also specify one or more initial values starting at a_1.

Solving a recurrence relation means finding a formula for a_n that does not involve any previous terms $a_1, \ldots, a_{n-1}$.

There are many methods of solving recurrence relations. Two important methods are **guess-and-check** and the **recursion tree method**.

We will make extensive use of the recursion tree method. However, we first take a quick look at the guess-and-check method.

Guess-and-check Method

step 1 Tabulate some values $a_1, a_2, \ldots$ using the recurrence relation.

step 2 Guess that the solution a_n has a specific form, involving undetermined constants.

step 3 Use $a_1, a_2, \ldots$ to determine specific values for the unspecified constants.

step 4 Use induction to prove your guess for a_n is correct.

Example of the Guess-and-check Method

Suppose we have the recurrence $T(n) = T(n-1) + 6n - 5$, $T(0) = 4$.

We compute a few values: $T(1) = 5$, $T(2) = 12$, $T(3) = 25$, $T(4) = 44$.

If we are sufficiently perspicacious, we might guess that $T(n)$ is a **quadratic function**, e.g., $T(n) = an^2 + bn + c$.

Next, we use $T(0) = 4$, $T(1) = 5$, $T(2) = 12$ to compute a, b and c by solving three equations in three unknowns.

We get $a = 3$, $b = -2$, $c = 4$.

Now we can use induction to prove that $T(n) = 3n^2 - 2n + 4$ for all $n \geq 0$.

Recursion Tree Method

The following recurrence relation arises in the analysis of *Mergesort*:

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \text{ is a power of } 2 \\ d & \text{if } n = 1, \end{cases}$$

where c and d are constants.

We can solve this recurrence relation when n is a power of two, by constructing a **recursion tree**, as follows:

step 1 Start with a one-node tree, say N, having the value $T(n)$.

step 2 Grow two children of N. These children, say N_1 and N_2, have the value $T(n/2)$, and the value of N is replaced by cn.

step 3 Repeat this process recursively, terminating when a node receives the value $T(1) = d$.

step 4 Sum the values on each level of the tree, and then compute the sum of all these sums; the result is $T(n)$.

Master Theorem

The Master Theorem provides a formula for the solution of many recurrence relations typically encountered in the analysis of algorithms.

The following is a simplified version of the Master Theorem:

Theorem

Suppose that $a \geq 1$ and $b > 1$. Consider the recurrence

$$T(n) = a\,T\left(\frac{n}{b}\right) + \Theta(n^y), \tag{1}$$

where n is a power of b. Denote $x = \log_b a$. Then

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } y < x \\ \Theta(n^x \log n) & \text{if } y = x \\ \Theta(n^y) & \text{if } y > x. \end{cases}$$

Proof of the Master Theorem (simplified version)

Suppose that $a \geq 1$ and $b \geq 2$ are integers and

$$T(n) = a\,T\left(\frac{n}{b}\right) + c\,n^y, \qquad T(1) = d.$$

Let $n = b^j$.

level	# nodes	value at each node	value of the level
j	1	$c\,n^y$	$c\,n^y$
$j-1$	a	$c\,(n/b)^y$	$c\,a\,(n/b)^y$
$j-2$	a^2	$c\,(n/b^2)^y$	$c\,a^2\,(n/b^2)^y$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
1	a^{j-1}	$c\,(n/b^{j-1})^y$	$c\,a^{j-1}\,(n/b^{j-1})^y$
0	a^j	d	$d\,a^j$

Computing $T(n)$

Summing the values at all levels of the recursion tree, we have that

$$T(n) = d\,a^j + c\,n^y \sum_{i=0}^{j-1} \left(\frac{a}{b^y}\right)^i.$$

Recall that $b^x = a$ and $n = b^j$. Hence $a^j = (b^x)^j = (b^j)^x = n^x$.

The formula for $T(n)$ is a **geometric sequence** with ratio $r = a/b^y = b^{x-y}$:

$$T(n) = d\,n^x + c\,n^y \sum_{i=0}^{j-1} r^i.$$

There are **three cases**, depending on whether $r > 1$, $r = 1$ or $r < 1$.

Complexity of $T(n)$

case	r	y, x	complexity of $T(n)$
heavy leaves	$r > 1$	$y < x$	$T(n) \in \Theta(n^x)$
balanced	$r = 1$	$y = x$	$T(n) \in \Theta(n^x \log n)$
heavy top	$r < 1$	$y > x$	$T(n) \in \Theta(n^y)$

heavy leaves means that the value of the recursion tree is dominated by the values of the leaf nodes.

balanced means that the values of the levels of the recursion tree are constant (except for the last level).

heavy top means that the value of the recursion tree is dominated by the value of the root node.

Master Theorem (modified general version)

Theorem

Suppose that $a \geq 1$ *and* $b > 1$. *Consider the recurrence*

$$T(n) = a\,T\left(\frac{n}{b}\right) + f(n),$$

where n *is a power of* b. *Denote* $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \text{if } f(n) \in O(n^{x-\epsilon}) \text{ for some } \epsilon > 0 \\ \Theta(n^x \log n) & \text{if } f(n) \in \Theta(n^x) \\ \Theta(f(n)) & \text{if } f(n)/n^{x+\epsilon} \text{ is an increasing function of } n \\ & \text{for some } \epsilon > 0. \end{cases}$$

The Divide-and-Conquer Design Strategy

divide: Given a problem instance I, construct one or more smaller problem instances, denoted $I_1, \ldots, I_a$ (these are called **subproblems**). Usually, we want the size of these subproblems to be small compared to the size of I, e.g., half the size.

conquer: For $1 \leq j \leq a$, solve instance I_j recursively, obtaining solutions $S_1, \ldots, S_a$.

combine: Given $S_1, \ldots, S_a$, use an appropriate **combining** function to find the solution S to the problem instance I, i.e., $S \leftarrow \mathit{Combine}(S_1, \ldots, S_a)$.

Example: Design of Mergesort

Here, a problem instance consists of an array A of n integers, which we want to sort in increasing order. The size of the problem instance is n.

divide: Split A into two subarrays: A_L consists of the first $\lceil \frac{n}{2} \rceil$ elements in A and A_R consists of the last $\lfloor \frac{n}{2} \rfloor$ elements in A.

conquer: Run *Mergesort* on A_L and A_R.

combine: After A_L and A_R have been sorted, use a function *Merge* to merge A_L and A_R into a single sorted array. Recall that this can be done in time $\Theta(n)$ with a single pass through A_L and A_R. We simply keep track of the "current" element of A_L and A_R, always copying the smaller one into the sorted array.

Mergesort

Algorithm: *Mergesort*$(A : array; n : integer)$

if $n = 1$

then $S \leftarrow A$

else $\begin{cases} n_L \leftarrow \lceil \frac{n}{2} \rceil \\ n_R \leftarrow \lfloor \frac{n}{2} \rfloor \\ A_L \leftarrow [A[1], \ldots, A[n_L]] \\ A_R \leftarrow [A[n_L + 1], \ldots, A[n]] \\ S_L \leftarrow \textit{Mergesort}(A_L, n_L) \\ S_R \leftarrow \textit{Mergesort}(A_R, n_R) \\ S \leftarrow \textit{Merge}(S_L, n_L, S_R, n_R) \end{cases}$

return (S, n)

Analysis of Mergesort

Let $T(n)$ denote the time to run *Mergesort* on an array of length n.

divide takes time $\Theta(n)$

conquer takes time $T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right)$

combine takes time $\Theta(n)$

Recurrence relation:

$$T(n) = \begin{cases} T\left(\lceil \frac{n}{2} \rceil\right) + T\left(\lfloor \frac{n}{2} \rfloor\right) + \Theta(n) & \text{if } n > 1 \\ \Theta(1) & \text{if } n = 1. \end{cases}$$

Sloppy and Exact Recurrence Relations

It is simpler to replace the $\Theta(n)$ term by cn, where c is an unspecified constant. The resulting recurrence relation is called the **exact recurrence**.

$$T(n) = \begin{cases} T\left(\left\lceil \frac{n}{2} \right\rceil\right) + T\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

If we then remove the floors and ceilings, we obtain the so-called **sloppy recurrence**:

$$T(n) = \begin{cases} 2\,T\left(\frac{n}{2}\right) + cn & \text{if } n > 1 \\ d & \text{if } n = 1. \end{cases}$$

The exact and sloppy recurrences are identical when n is a power of two.

Further, the sloppy recurrence makes sense only when n is a power of two.

Complexity of the Solution to the Exact Recurrence

The Master Theorem provides the **exact** solution of the recurrence when $n = 2^j$ (it is in fact a **proof** for these values of n).

We can express this solution (for powers of 2) as a function of n, using Θ-notation.

It can be shown that the resulting function of n will in fact yield the **complexity** of the solution of the exact recurrence for **all values** of n.

This derivation of the complexity of $T(n)$ is not a proof, however. If a rigourous mathematical proof is required, then it is necessary to use **induction** along with the **exact recurrence**.

The Max-Min Problem

Let's design a divide-and-conquer algorithm for the **Max-Min** problem.

Divide: Suppose we split A into two equal-sized subarrays, A_L and A_R.

Conquer: We find the maximum and minimum elements in each subarray recursively, obtaining max_L, min_L, max_R and min_R.

Combine: Then we can easily "combine" the solutions to the two subproblems to solve the original problem instance:

$$max \leftarrow \max\{max_L, max_R\}$$

and

$$min \leftarrow \min\{min_L, min_R\}$$

The Max-Min Problem (cont.)

The recurrence relation describing the complexity of the running time is $T(n) = 2T(n/2) + \Theta(1)$.

The Master Theorem shows that the $T(n) \in \Theta(n)$.

However, we can also count the **exact number** of comparisons done by the algorithm, obtaining the (sloppy) recurrence

$$C(n) = 2C(n/2) + 2, \quad C(2) = 1.$$

For n a power of 2, the solution to this recurrence relation is $C(n) = 3n/2 - 2$, so the divide-and-conquer algorithm is **optimal** for these values of n (see slide 26).

Non-dominated Points

Given two points $(x_1, y_1), (x_2, y_2)$ in the Euclidean plane, we say that (x_1, y_1) **dominates** (x_2, y_2) if $x_1 \geq x_2$ and $y_1 \geq y_2$.

Problem

Non-dominated Points

Instance: *A list S of n points in the Euclidean plane, say $(x_1, y_1), \ldots, (x_n, y_n)$.*

Question: *Find all the* **non-dominated points** *in S, i.e., all the points that are not dominated by any other point in S.*

Non-dominated Points has a trivial $\Theta(n^2)$ algorithm to solve it, based on comparing all pairs of points in S. Can we do better?

Problem Decomposition

Observe that the non-dominated points form a **staircase** such that all the other points are "under" the staircase.

Suppose we **pre-sort** the points in S with respect to their x-co-ordinates. This takes time $\Theta(n \log n)$.

Divide: Let the first $n/2$ points be denoted S_1 and let the last $n/2$ points be denoted S_2.

Conquer: Recursively solve the subproblems defined by the two instances S_1 and S_2.

Combine: Given the non-dominated points in S_1 and the non-dominated points in S_2, how do we find the non-dominated points in S?

Observe that **no point in S_1 dominates a point in S_2**.

Therefore we only need to eliminate the points in S_1 that are dominated by a point in S_2. This can be done in time $O(n)$.

Closest Pair

Problem

Closest Pair

Instance: *a set* Q *of* n *distinct points in the Euclidean plane,*

$$Q = \{Q[1], \ldots, Q[n]\}.$$

Find: *Two distinct points* $Q[i] = (x, y), Q[j] = (x', y')$ *such that the Euclidean distance*

$$\sqrt{(x' - x)^2 + (y' - y)^2}$$

is minimized.

Closest Pair: Problem Decomposition

Suppose we presort the points in Q with respect to their x-coordinates (this takes time $\Theta(n \log n)$).

Then we can easily find the vertical line that partitions the set of points Q into two sets of size $n/2$: this line has equation $x = Q[m].x$, where $m = n/2$.

The set Q is global with respect to the recursive procedure *ClosestPair1*.

At any given point in the recursion, we are examining a subarray $(Q[\ell], \ldots, Q[r])$, and $m = \lfloor (\ell + r)/2 \rfloor$.

We call *ClosestPair1*$(1, n)$ to solve the given problem instance.

Closest Pair: Solution 1

Algorithm: *ClosestPair1*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else $\begin{cases} m \leftarrow \lfloor(\ell + r)/2\rfloor \\ \delta_L \leftarrow \textit{ClosestPair1}(\ell, m) \\ \delta_R \leftarrow \textit{ClosestPair1}(m+1, r) \\ \delta \leftarrow \min\{\delta_L, \delta_R\} \\ R \leftarrow \textit{SelectCandidates}(\ell, r, \delta, Q[m].x) \\ R \leftarrow \textit{SortY}(R) \\ \delta \leftarrow \textit{CheckStrip}(R, \delta) \end{cases}$

return (δ)

Selecting Candidates from the Vertical Strip

Algorithm: *SelectCandidates*$(\ell, r, \delta, xmid)$

$j \leftarrow 0$

for $i \leftarrow \ell$ **to** r

$$\textbf{do} \begin{cases} \textbf{if } |Q[i].x - xmid| \leq \delta \\ \quad \textbf{then } \begin{cases} j \leftarrow j + 1 \\ R[j] \leftarrow Q[i] \end{cases} \end{cases}$$

return (R)

Checking the Vertical Strip

Algorithm: *CheckStrip*(R, δ)

$t \leftarrow$ *size*(R)
$\delta' \leftarrow \delta$
for $j \leftarrow 1$ **to** $t - 1$

$$\textbf{do} \begin{cases} \textbf{for } k \leftarrow j+1 \textbf{ to } \min\{t, j+7\} \\ \quad \textbf{do} \begin{cases} x \leftarrow R[j].x \\ x' \leftarrow R[k].x \\ y \leftarrow R[j].y \\ y' \leftarrow R[k].y \\ \delta' \leftarrow \min\left\{\delta', \sqrt{(x'-x)^2 + (y'-y)^2}\right\} \end{cases} \end{cases}$$

return (δ')

Closest Pair: Solution 2

Algorithm: *ClosestPair2*(ℓ, r)

if $\ell = r$ **then** $\delta \leftarrow \infty$

else $\begin{cases} m \leftarrow \lfloor (\ell + r)/2 \rfloor \\ \delta_L \leftarrow \textit{ClosestPair2}(\ell, m) \\ \text{comment: } Q[\ell], \ldots, Q[m] \text{ is sorted WRT } y\text{-coordinates} \\ \delta_R \leftarrow \textit{ClosestPair2}(m+1, r) \\ \text{comment: } Q[m+1], \ldots, Q[r] \text{ is sorted WRT } y\text{-coordinates} \\ \delta \leftarrow \min\{\delta_L, \delta_R\} \\ \textit{Merge}(\ell, m, r) \\ R \leftarrow \textit{SelectCandidates}(\ell, r, \delta, Q[m].x) \\ \delta \leftarrow \textit{CheckStrip}(R, \delta) \end{cases}$

return (δ)

Multiprecision Multiplication

Problem

Multiprecision Multiplication

Instance: *Two k-bit positive integers, X and Y, having binary representations*

$$X = [X[k-1], ..., X[0]]$$

and

$$Y = [Y[k-1], ..., Y[0]].$$

Question: *Compute the $2k$-bit positive integer $Z = XY$, where*

$$Z = (Z[2k-1], ..., Z[0]).$$

We are interested in the **bit complexity** of algorithms that solve **Multiprecision Multiplication**, which means that the complexity is expressed as a function of k (the size of the problem instance is $2k$ bits).

Not-So-Fast D&C Multiprecision Multiplication

Algorithm: *NotSoFastMultiply*(X, Y, k)
if $k = 1$
then $Z \leftarrow X[0] \times Y[0]$
else $\begin{cases} Z_1 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{NotSoFastMultiply}(X_L, Y_R, k/2) \\ Z_4 \leftarrow \textit{NotSoFastMultiply}(X_R, Y_L, k/2) \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + Z_2 + \textit{LeftShift}(Z_3 + Z_4, k/2) \end{cases}$
return (Z)

Fast D&C Multiprecision Multiplication

Algorithm: *FastMultiply*(X, Y, k)

if $k = 1$

then $Z \leftarrow X[0] \times Y[0]$

else $\begin{cases} X_T \leftarrow X_L + X_R \\ Y_T \leftarrow Y_L + Y_R \\ Z_1 \leftarrow \textit{FastMultiply}(X_L, Y_L, k/2) \\ Z_2 \leftarrow \textit{FastMultiply}(X_R, Y_R, k/2) \\ Z_3 \leftarrow \textit{FastMultiply}(X_T, Y_T, k/2), \\ Z \leftarrow \textit{LeftShift}(Z_1, k) + Z_2 + \textit{LeftShift}(Z_3 - Z_1 - Z_2, k/2) \end{cases}$

return (Z)

Matrix Multiplication

Problem

Matrix Multiplication

Instance: *Two n by n matrices, A and B.*

Question: *Compute the n by n matrix product $C = AB$.*

The naive algorithm for **Matrix Multiplication** has complexity $\Theta(n^3)$.

Matrix Multiplication: Problem Decomposition

$$A = \begin{pmatrix} a & b \\ c & d \end{pmatrix}, \quad B = \begin{pmatrix} e & f \\ g & h \end{pmatrix}, \quad C = AB = \begin{pmatrix} r & s \\ t & u \end{pmatrix}$$

If A, B are n by n matrices, then $a, b, ..., h, r, s, t, u$ are $\frac{n}{2}$ by $\frac{n}{2}$ matrices, where

$$\begin{aligned} r &= a\,e + b\,g & \qquad s &= a\,f + b\,h \\ t &= c\,e + d\,g & \qquad u &= c\,f + d\,h \end{aligned}$$

We require 8 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

Efficient D&C Matrix Multiplication

Define

$$
\begin{aligned}
P_1 &= a(f-h) & P_2 &= (a+b)h \\
P_3 &= (c+d)e & P_4 &= d(g-e) \\
P_5 &= (a+d)(e+h) & P_6 &= (b-d)(g+h) \\
P_7 &= (a-c)(e+f). & &
\end{aligned}
$$

Then, compute

$$
\begin{aligned}
r &= P_5 + P_4 - P_2 + P_6 & s &= P_1 + P_2 \\
t &= P_3 + P_4 & u &= P_5 + P_1 - P_3 - P_7.
\end{aligned}
$$

We now require only 7 multiplications of $\frac{n}{2}$ by $\frac{n}{2}$ matrices in order to compute $C = AB$.

Selection

Problem

Selection

Instance: *An array* $A[1], \ldots, A[n]$ *of distinct integer values, and an integer* k*, where* $1 \leq k \leq n$*.*

Find: *The* k*th smallest integer in the array* A*.*

The problem **Median** is the special case of **Selection** where $k = \lceil \frac{n}{2} \rceil$.

QuickSelect

Suppose we choose a **pivot** element y in the array A, and we **restructure** A so that all elements less than y precede y in A, and all elements greater than y occur after y in A. (This is exactly what is done in *Quicksort*, and it takes **linear time**.)

Suppose that $A[posn] = y$ after restructuring. Let A_L be the subarray $A[1], \ldots, A[posn-1]$ and let A_R be the subarray (of size $n - posn$) $A[posn+1], \ldots, A[n]$.

Then the kth smallest element of A is

$$\begin{cases} y & \text{if } k = posn \\ \text{the } k\text{th smallest element of } A_L & \text{if } k < posn \\ \text{the } (k-posn)\text{th smallest element of } A_R & \text{if } k > posn. \end{cases}$$

We make (at most) one recursive call at each level of the recursion.

Average-case Analysis of QuickSelect

We say that a pivot is **good** if $posn$ is in the middle half of A.

The probability that a pivot is good is $1/2$.

On average, after **two iterations**, we will encounter a good pivot.

If a pivot is good, then $|A_L| \leq 3n/4$ and $|A_R| \leq 3n/4$.

With an **expected** linear amount of work, the size of the subproblem is reduced by at least 25%.

It follows that the average-case complexity of the *QuickSelect* is linear.

Achieving $O(n)$ Worst-Case Complexity: A Strategy for Choosing the Pivot

We choose the pivot to be a certain **median-of-medians**:

step 1 Given $n \geq 15$, write $n = 10r + 5 + \theta$, where $r \geq 1$ and $0 \leq \theta \leq 9$.

step 2 Divide A into $2r + 1$ disjoint subarrays of 5 elements. Denote these subarrays by $B_1, \ldots, B_{2r+1}$.

step 3 For $1 \leq i \leq 2r + 1$, find the median of B_i (nonrecursively), and denote it by m_i.

step 4 Define M to be the array consisting of elements $m_1, \ldots, m_{2r+1}$.

step 5 Find the median y of the array M (recursively).

step 6 Use the element y as the pivot for A.

Median-of-medians-QuickSelect

Algorithm: *Mom-QuickSelect*(k, n, A)

if $n \leq 14$ **then** sort A and **return** $(A[k])$
write $n = 10r + 5 + \theta$, where $0 \leq \theta \leq 9$
construct $B_1, \ldots, B_{2r+1}$ (subarrays of A, each of size 5)
find medians $m_1, \ldots, m_{2r+1}$ (non-recursively)
$M \leftarrow [m_1, \ldots, m_{2r+1}]$
$y \leftarrow$ *Mom-QuickSelect*$(r + 1, 2r + 1, M)$
$(A_L, A_R, posn) \leftarrow$ *Restructure*(A, y)
if $k = posn$ **then return** (y)
  else if $k < posn$ **then return** (*Mom-QuickSelect*$(k, posn - 1, A_L)$)
   else return (*Mom-QuickSelect*$(k - posn, n - posn, A_R)$)

Worst-case Analysis of Mom-QuickSelect

When the pivot is the median-of-medians, we have that $|A_L| \leq \left\lfloor \frac{7n+12}{10} \right\rfloor$ and $A_R \leq \left\lfloor \frac{7n+12}{10} \right\rfloor$.

The *Mom-QuickSelect* algorithm requires **two recursive calls**.

The worst-case complexity $T(n)$ of this algorithm satisfies the following recurrence:

$$T(n) \leq \begin{cases} T\left(\left\lfloor \frac{n}{5} \right\rfloor\right) + T\left(\left\lfloor \frac{7n+12}{10} \right\rfloor\right) + \Theta(n) & \text{if } n \geq 15 \\ \Theta(1) & \text{if } n \leq 14. \end{cases}$$

It can be shown that $T(n)$ is $O(n)$.