CS 341: Algorithms
Module 3: Reductions, Recurrences

Armin Jamshidpey, Eugene Zima

Based on lecture notes by many previous CS 341 instructors

David R. Cheriton School of Computer Science, University of Waterloo

Spring 2019

The 2SUM problem

Problem:

Given an array A[1..n] of integers and integer m, find if there are indices i and j (not necessarily distinct!) such that $A[i]+A[j] = m$.

First solution

Algorithm 1: $\textsc{Simple2SUM}(A[1..n], m)$

```
for i = 1 to n do
    for j = i to n do
        if A[i]+A[j] = m then
            return true
return false
```

Analyze this ... $\Theta(n^2)$

Second solution

Algorithm 2: $\textsc{Faster2SUM}(A[1..n],\ m)$

Sort(A)
for $i = 1$ **to** n **do**
    j = BinarySearch(m-A[i], A)
    if $j > 0$ **then**
        return *true*
return *false*

Analyze this ... $\Theta(n \log n)$

Third solution (Assuming array is already sorted by Sort(A));

Algorithm 3: $\textsc{Sorted2SUM}(A[1..n],\ m)$

```
i = 1; j = n;
while i ≤ j do
    if A[i] + A[j] = m then
        return true
    else
        if A[i] + A[j] < m then
            i = i+1
        else
            j = j-1
return false
```

Analyze this ... $\Theta(n \log n)$, but the second stage ($\textsc{Sorted2SUM}$) is $\Theta(n)$.

Sketch of correctness proof:

If there is no such pair of indices, our algorithm will not find it (as we will never get *true* at line 3).

Sketch of correctness proof:

If there is no such pair of indices, our algorithm will not find it (as we will never get *true* at line 3).

Suppose there is such pair $i' \leq j'$ such that $A[i'] + A[j'] = m$, we only need to prove that our algorithm will not miss this pair. Without loss of generality we suppose i becomes i' when $j > j'$. The other direction can be proved by symmetry.

Sketch of correctness proof:

If there is no such pair of indices, our algorithm will not find it (as we will never get *true* at line 3).

Suppose there is such pair $i' \leq j'$ such that $A[i'] + A[j'] = m$, we only need to prove that our algorithm will not miss this pair. Without loss of generality we suppose i becomes i' when $j > j'$. The other direction can be proved by symmetry.

Since array is sorted, $A[j] \geq A[j']$. So, the if-else branch will not increase i anymore. It will keep reducing j until finds j' pair.

The 3SUM problem

Problem:

Given an integer array $A[1..n]$, find i, j and k such that

$$A[i]+A[j]+A[k] = 0.$$

The 3SUM problem

Problem:

Given an integer array $A[1..n]$, find i, j and k such that

$$A[i]+A[j]+A[k] = 0.$$

Solution 1:

Three nested loops to check each triplet (i, j, k) ($\Theta(n^3)$).

The 3SUM problem

Problem:

Given an integer array A[1..n], find i, j and k such that

$$\mathrm{A}[i]+\mathrm{A}[j]+\mathrm{A}[k] = 0.$$

Solution 1:

Three nested loops to check each triplet (i, j, k) ($\Theta(n^3)$).

Solution 2:

For each value of k, use FASTER2SUM (second solution) to find $i, j : A[i] + A[j] = -A[k]$ ($\Theta(n^2 \log n)$).

The 3SUM problem

Problem:

Given an integer array A[1..n], find i, j and k such that

$$\mathrm{A}[i]+\mathrm{A}[j]+\mathrm{A}[k] = 0.$$

Solution 1:

Three nested loops to check each triplet (i, j, k) ($\Theta(n^3)$).

Solution 2:

For each value of k, use FASTER2SUM (second solution) to find $i, j : A[i] + A[j] = -A[k]$ ($\Theta(n^2 \log n)$).

This is the reduction technique in algorithm design.

The 3SUM problem

Solution 3:

Pre-sort A to avoid sorting it for each k, still use $\textsc{Faster2SUM}$ idea (binary search).

The 3SUM problem

Solution 3:

Pre-sort A to avoid sorting it for each k, still use FASTER2SUM idea (binary search).

Solution 4: Use a better SORTED2SUM solution (on array sorted once)!

Algorithm 5: FAST3SUM($A[1..n]$, m)

```
Sort(A);
for k = 1 to n do
    x = Sorted2SUM(A,-A[k])
    if x then
        return true
return false
```

Analyze this ... $\Theta(n^2)$

Recurrence Relations

The mergesort recurrence is

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

Recurrence Relations

The mergesort recurrence is

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

It is simpler to consider the following *exact* recurrence, with constant factors c and d replacing Θ's:

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

Recurrence Relations (cont.)

The following is the corresponding *sloppy* recurrence (it has floors and ceilings removed):

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

Recurrence Relations (cont.)

The following is the corresponding *sloppy* recurrence (it has floors and ceilings removed):

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

The exact and sloppy recurrences are identical when n is a power of 2.

Recurrence Relations (cont.)

The following is the corresponding *sloppy* recurrence (it has floors and ceilings removed):

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

The exact and sloppy recurrences are identical when n is a power of 2.

We will begin by solving the sloppy recurrence when $n = 2^j$ using the *recursion tree method*.

Recursion Tree Method

We can construct a recursion tree for the sloppy recurrence, assuming $n = 2^j$, as follows.

Recursion Tree Method

We can construct a recursion tree for the sloppy recurrence, assuming $n = 2^j$, as follows.

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

Recursion Tree Method

We can construct a recursion tree for the sloppy recurrence, assuming $n = 2^j$, as follows.

1. Start with a one-node tree, say N, which receives the value $T(n)$.
2. Grow two children of N. These children, say N_1 and N_2, receive the value $T(n/2)$, and the value of N is updated to be cn.

Recursion Tree Method

We can construct a recursion tree for the sloppy recurrence, assuming $n = 2^j$, as follows.

1. Start with a one-node tree, say N, which receives the value $T(n)$.
2. Grow two children of N. These children, say N_1 and N_2, receive the value $T(n/2)$, and the value of N is updated to be cn.
3. Repeat this process recursively, terminating when a node receives the value $T(1) = d$.

Recursion Tree Method

We can construct a recursion tree for the sloppy recurrence, assuming $n = 2^j$, as follows.

1. Start with a one-node tree, say N, which receives the value $T(n)$.
2. Grow two children of N. These children, say N_1 and N_2, receive the value $T(n/2)$, and the value of N is updated to be cn.
3. Repeat this process recursively, terminating when a node receives the value $T(1) = d$.
4. Sum the values on each level of the tree, and then compute the sum of all these sums; the result is $T(n)$.

Solving the Exact Recurrence

- The recursion tree method finds the solution of the exact recurrence when $n = 2^j$ (it is in fact a proof for these values of n).

Solving the Exact Recurrence

- The recursion tree method finds the solution of the exact recurrence when $n = 2^j$ (it is in fact a proof for these values of n).
- If this solution is expressed as a function of n using Θ-notation, then we obtain the complexity of the solution of the exact recurrence for *all* n.

Solving the Exact Recurrence

- The recursion tree method finds the solution of the exact recurrence when $n = 2^j$ (it is in fact a proof for these values of n).
- If this solution is expressed as a function of n using Θ-notation, then we obtain the complexity of the solution of the exact recurrence for *all* n.
- This is not a proof, however. If a real mathematical proof is required, then it is necessary to use induction.

The Master Method

The "Master Theorem" provides a formula for the solution of many recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

The Master Method

The “Master Theorem” provides a formula for the solution of many recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

The following is a simplified version (a more general version can be found in the textbook):

The Master Method

The "Master Theorem" provides a formula for the solution of many recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

The following is a simplified version (a more general version can be found in the textbook):

Theorem (Master theorem)

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

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

in sloppy or exact form. Denote $x = \log_b a$. Then

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

The Master Method

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 $n = b^j$.

The Master Method

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 $n = b^j$.

size of subproblem	# nodes	cost/node	total cost
$n = b^j$	1	$c\,n^y$	$c\,n^y$
$n/b = b^{j-1}$	a	$c\,(n/b)^y$	$c\,a\,(n/b)^y$
$n/b^2 = b^{j-2}$	a^2	$c\,(n/b^2)^y$	$c\,a^2\,(n/b^2)^y$
$\vdots$	$\vdots$	$\vdots$	$\vdots$
$n/b^{j-1} = b$	a^{j-1}	$c\,(n/b^{j-1})^y$	$c\,a^{j-1}\,(n/b^{j-1})^y$
$n/b^j = 1$	a^j	d	$d\,a^j$

Computing $T(n)$

Summing the costs of all levels of the recursion tree, we have that

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

Computing $T(n)$

Summing the costs of 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 $a^j = (b^x)^j = (b^j)^x = n^x$.

Computing $T(n)$

Summing the costs of 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 $r = \frac{a}{b^y} = b^{x-y}$:

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

Computing $T(n)$

Summing the costs of 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 $r = \frac{a}{b^y} = b^{x-y}$:

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

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	$r < 1$	$y > x$	$T(n) \in \Theta(n^y)$

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 cost of the recursion tree is dominated by the cost of the leaf nodes.

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 cost of the recursion tree is dominated by the cost of the leaf nodes.

"balanced" means that costs of the levels of the recursion tree stay "almost the same" (except for the last level)

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 cost of the recursion tree is dominated by the cost of the leaf nodes.

"balanced" means that costs of the levels of the recursion tree stay "almost the same" (except for the last level)

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

Master Theorem (slightly more general version)

This version provides a formula for even more recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

Master Theorem (slightly more general version)

This version provides a formula for even more recurrence relations typically encountered in the analysis of divide-and-conquer algorithms.

Theorem

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

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

Denote $x = \log_b a$. *Then*

$$T(n) \in \begin{cases} \Theta(n^x) & \textit{if } y < x \\ \Theta(n^y \log^{m+1} n) & \textit{if } y = x \\ \Theta(n^y \log^m n) & \textit{if } y > x. \end{cases}$$