Divide and Conquer Algorithms

CSE 101: Design and Analysis of Algorithms
Lecture 12
CSE 101: Design and analysis of algorithms

• Divide and conquer algorithms
  – Reading: Sections 2.1 and 2.2

• Homework 5 is due today, 11:59 PM

• Quiz 2 is Nov 8, in class
  – Graph search and minimum spanning trees

• Homework 6 will be assigned Nov 13
  – No homework this week
Divide and conquer

• Break a problem into similar subproblems
• Solve each subproblem recursively
• Combine
Multiplying binary numbers, divide and conquer

• Suppose we want to multiply two n-bit numbers together where n is a power of 2
• One way we can do this is by splitting each number into their left and right halves which are each n/2 bits long

• $x = x_L x_R$
• $y = y_L y_R$
Multiplying binary numbers, divide and conquer

• Suppose we want to multiply two n-bit numbers together where n is a power of 2
• One way we can do this is by splitting each number into their left and right halves which are each n/2 bits long

\[
x = 2^{n/2}x_L + x_R
\]
\[
y = 2^{n/2}y_L + y_R
\]
Multiplying binary numbers, divide and conquer

\[ x = 2^{n/2} x_L + x_R \]
\[ y = 2^{n/2} y_L + y_R \]

\[ xy = \left(2^{\frac{n}{2}} x_L + x_R\right)\left(2^{\frac{n}{2}} y_L + y_R\right) \]
\[ xy = 2^n x_L y_L + 2^{\frac{n}{2}} (x_L y_R + x_R y_L) + x_R y_R \]

4 multiplications
Algorithm multiply

function multiply(x, y)  T(n)
Input: n-bit integers x and y
Output: the product xy
If n=1: return xy

\(x_L, x_R\) and \(y_L, y_R\) are the left-most and right-most \(n/2\) bits of \(x\) and \(y\), respectively.

\[P_1 = \text{multiply}(x_L, y_L)\quad T(n/2)\]
\[P_2 = \text{multiply}(x_L, y_R)\quad T(n/2)\]
\[P_3 = \text{multiply}(x_R, y_L)\quad T(n/2)\]
\[P_4 = \text{multiply}(x_R, y_R)\quad T(n/2)\]

return \(P_1 \cdot 2^n + (P_2 + P_3) \cdot 2^{n/2} + P_4\)
Algorithm multiply runtime

• Let $T(n)$ be the runtime of the multiply algorithm

• Then, $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$

Non-recursive part
Algorithm multiplyKS

function multiplyKS(x,y) \( T(n) \)
Input: n-bit integers x and y
Output: the product xy
If n=1: return xy
\( x_L, x_R \) and \( y_L, y_R \) are the left-most and right-most n/2 bits of x and y, respectively.
\[
\begin{align*}
R_1 &= \text{multiplyKS}(x_L, y_L) \quad T(n/2) \\
R_2 &= \text{multiplyKS}(x_R, y_R) \quad T(n/2) \\
R_3 &= \text{multiplyKS}((x_L + x_R)(y_L + y_R)) \quad T(n/2)
\end{align*}
\]
return \( R_1 * 2^n + (R_3 - R_1 - R_2) * 2^{n/2} + R_2 \)
Algorithm multiplyKS runtime

• Let $T(n)$ be the runtime of the multiplyKS algorithm

• Then, $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$
Recurrence of the same form

- multiply runtime $T(n) = 4T\left(\frac{n}{2}\right) + O(n)$
- multiplyKS runtime $T(n) = 3T\left(\frac{n}{2}\right) + O(n)$
Recurrence of the same form

• multiply runtime \( T(n) = 4T\left(\frac{n}{2}\right) + O(n) \)

• multiplyKS runtime \( T(n) = 3T\left(\frac{n}{2}\right) + O(n) \)

• How do you solve a recurrence of the form

\[
T(n) = aT\left(\frac{n}{b}\right) + O(n^d)
\]

• We will use the master theorem
Summation lemma

• Consider the summation

\[ \sum_{k=0}^{n} r^k \]

Geometric series

• It behaves differently for different values of \( r \)
Summation lemma

• Consider the summation

\[ \sum_{k=0}^{n} r^k \]

• It behaves differently for different values of \( r \)
• What happens if \( r < 1 \)?
Summation lemma

• Consider the summation

\[ \sum_{k=0}^{n} r^k \]

• It behaves differently for different values of \( r \)
• If \( r < 1 \), then this sum converges. This means that the sum is bounded above by some constant \( c \). Therefore, if \( r < 1 \), then \( \sum_{k=0}^{n} r^k < c \) for all \( n \), so \( \sum_{k=0}^{n} r^k \in O(1) \)
Summation lemma

• Consider the summation

\[ \sum_{k=0}^{n} r^k \]

• It behaves differently for different values of \( r \)

• What happens if \( r = 1 \)?
Summation lemma

• Consider the summation
\[ \sum_{k=0}^{n} r^k \]

• It behaves differently for different values of \( r \)

• If \( r = 1 \), then this sum is just summing 1 over and over \( n + 1 \) times. Therefore,
  
  if \( r = 1 \), then \( \sum_{k=0}^{n} r^k = \sum_{k=0}^{n} 1 = n + 1 \in O(n) \)
Summation lemma

• Consider the summation

\[ \sum_{k=0}^{n} r^k \]

• It behaves differently for different values of \( r \)

• What happens if \( r > 1 \)?
Summation lemma

• Consider the summation
\[ \sum_{k=0}^{n} r^k \]

• It behaves differently for different values of \( r \)

• If \( r > 1 \), then this sum is exponential with base \( r \).
Therefore,
if \( r > 1 \), then \( \sum_{k=0}^{n} r^k < cr^n \) for all \( n \), so \( \sum_{k=0}^{n} r^k \in O(r^n) \)
(note that \( c > \frac{r}{r-1} \))
Summation lemma

• Consider the summation

\[ \sum_{k=0}^{n} r^k \]

• It behaves differently for different values of \( r \)

\[ \sum_{k=0}^{n} r^k \in \begin{cases} 
O(1) & \text{if } r < 1 \\
O(n) & \text{if } r = 1 \\
O(r^n) & \text{if } r > 1 
\end{cases} \]
Master theorem

• If $T(n) = aT(n/b) + O(n^d)$ for some constants $a > 0, b > 1, d \geq 0$, then

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

• The master theorem tell us us the running times of most of the divide and conquer procedures we are likely to use
Master theorem, solving the recurrence

\[ T(n) = aT\left(\frac{n}{b}\right) + O(n^d) \]

1 subproblem

\( a \) subproblems

\( a^2 \) subproblems

\( a^{\log_b n} \) subproblems

CSE 101, Fall 2018
Master theorem, solving the recurrence

\[ T(n) = aT\left(\frac{n}{b}\right) + O(n^d) \]

- 1 subproblem
- \(a\) subproblems
- \(a^2\) subproblems
- \(a^{\log_b n}\) subproblems

CSE 101, Fall 2018
Time analysis, multiply

Adding n bit numbers = cn time

- $n/2$ bit adds = $cn/2$
- $n/2$ bit adds = $cn/2$
- $n/2$ bit adds = $cn/2$
- $n/2$ bit adds = $cn/2$

Total 1st level down: $4 \times cn/2 = 2cn$

Total 2nd level down: $16 \times cn/4 = 4cn$
Adding $n$ bit numbers = $cn$ time

- $n/2$ bit adds = $cn/2$
- $n/2$ bit adds = $cn/2$
- $n/2$ bit adds = $cn/2$

Total 1\textsuperscript{st} level down: $3 \times cn/2 = (1.5) \times cn$

Total 2\textsuperscript{nd} level down: $9 \times cn/4 = (1.5)^2 \times cn$
Master theorem, solving the recurrence

- After \( k \) levels, there are \( a^k \) subproblems, each of size \( n/b^k \)
- So, during the \( k \)th level of recursion, the time complexity is

\[
O\left(\left(\frac{n}{b^k}\right)^d\right)a^k = O\left(a^k\left(\frac{n}{b^k}\right)^d\right) = O\left(n^d\left(\frac{a}{b^d}\right)^k\right)
\]
Master theorem, solving the recurrence

- After \( k \) levels, there are \( a^k \) subproblems, each of size \( n/b^k \)
- So, during the \( k \)th level of recursion, the time complexity is

\[
O\left(\left(\frac{n}{b^k}\right)^a\right) a^k = O\left(a^k \left(\frac{n}{b^k}\right)^a\right)
\]

\[
= O\left(n^a \left(\frac{a}{b^d}\right)^k\right)
\]
Master theorem, solving the recurrence

• After $k$ levels, there are $a^k$ subproblems, each of size $n/b^k$

• So, during the $k$th level of recursion, the time complexity is

$$O \left( \left( \frac{n}{b^k} \right)^d \right)^{a^k} = O \left( a^k \left( \frac{n}{b^k} \right)^d \right)$$

$$= O \left( n^d \left( \frac{a}{b^d} \right)^k \right)$$

• After $\log_b n$ levels, the subproblem size is reduced to 1, which usually is the size of the base case

• So, the entire algorithm is a sum of each level

$$T(n) = O \left( n^d \sum_{k=0}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right)$$
Master theorem, solving the recurrence

- After $k$ levels, there are $a^k$ subproblems, each of size $n/b^k$
- So, during the $k$th level of recursion, the time complexity is

$$O \left( \left( \frac{n}{b^k} \right)^d \right) a^k = O \left( a^k \left( \frac{n}{b^k} \right)^d \right)$$

$$= O \left( n^d \left( \frac{a}{b^d} \right)^k \right)$$

- After $\log_b n$ levels, the subproblem size is reduced to 1, which usually is the size of the base case
- So, the entire algorithm is a sum of each level

$$T(n) = O \left( n^d \sum_{k=0}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right)$$
Master theorem, proof

\[ T(n) = O \left( n^d \sum_{k=0}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right) \]

- Case 1: \( a < b^d \)

Then, \( \frac{a}{b^d} < 1 \) and the series converges to a constant so

\[ T(n) = O(n^d) \]
Master theorem, proof

\[ T(n) = O \left( n^d \sum_{k=0}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right) \]

- Case 1: \( a < b^d \)
  - Then, \( \frac{a}{b^d} < 1 \) and the series converges to a constant so
  \[ T(n) = O(n^d) \]
Master theorem, proof

\[
T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right)
\]

• Case 2: \(a = b^d\)

• Then, \(\frac{a}{b^d} = 1\) and so each term is equal to 1 so

\[
T(n) = O\left(n^d \log_b n\right)
\]
Master theorem, proof

\[ T(n) = O \left( n^d \sum_{k=0}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right) \]

- Case 2: \( a = b^d \)
- Then, \( \frac{a}{b^d} = 1 \) and so each term is equal to 1 so

\[ T(n) = O \left( n^d \log_b n \right) \]
Master theorem, proof

\[ T(n) = O \left( n^d \sum_{k=0}^{\log_b n} \left( \frac{a}{b^d} \right)^k \right) \]

- Case 3: \( a > b^d \)
- Then, the summation is exponential and grows proportional to its last term \( \left( \frac{a}{b^d} \right)^{\log_b n} \)
  
  so

\[ T(n) = O \left( n^d \left( \frac{a}{b^d} \right)^{\log_b n} \right) = O(n^{\log_b a}) \]
Master theorem, proof

\[ T(n) = O\left(n^d \sum_{k=0}^{\log_b n} \left(\frac{a}{b^d}\right)^k\right) \]

- Case 3: \( a > b^d \)
- Then, the summation is exponential and grows proportional to its last term \( \left(\frac{a}{b^d}\right)^{\log_b n} \) so

\[ T(n) = O\left(n^d \left(\frac{a}{b^d}\right)^{\log_b n}\right) = O(n \log_b a) \]  

Exercise
Master theorem

• If $T(n) = aT(n/b) + O(n^d)$ for some constants $a > 0$, $b > 1$, $d \geq 0$, then

$$T(n) \in \begin{cases} 
O(n^d) & \text{if } a < b^d \\
O(n^d \log n) & \text{if } a = b^d \\
O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases}$$
Master theorem applied to multiply

• The recursion for the runtime of multiply is

\[ T(n) = 4T\left(\frac{n}{2}\right) + O(n) \]

• So, we have that \( a = 4, b = 2, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in O\left(n^{\log_4 4}\right) = O(n^2) \]

• Not any improvement on grade-school method
Master theorem applied to multiply

• The recursion for the runtime of multiply is

\[ T(n) = 4T \left( \frac{n}{2} \right) + \Theta(n) \]

• So, we have that \( a = 4, b = 2, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in O(n^{\log_2 4}) = O(n^2) \]

• Not any improvement on grade-school method
Master theorem applied to multiplyKS

• The recursion for the runtime of multiplyKS is

\[ T(n) = 3T\left(\frac{n}{2}\right) + O(n) \]

• So, we have that \( a = 3, \ b = 2, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in O(n^{\log_2 3}) = O(n^{1.58}) \]

• An improvement on grade-school method
Master theorem applied to multiplyKS

• The recursion for the runtime of multiplyKS is

\[ T(n) = 3T\left(\frac{n}{2}\right) + \Theta(n) \]

• So, we have that \( a = 3, b = 2, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in \Theta(n^{\log_2 3}) = \Theta(n^{1.58}) \]

• An improvement on grade-school method
Can we do better than $n^{1.58}$?

• Could any multiplication algorithm have a faster asymptotic runtime than $\Theta(n^{1.58})$?

• Ideas
Can we do better than $n^{1.58}$?

- Instead of splitting the numbers in half, we split them into thirds

- $x = x_L \quad x_M \quad x_R$
- $y = y_L \quad y_M \quad y_R$
Can we do better than $n^{1.58}$?

- Instead of splitting the numbers in half, we split them into thirds
  
  - $x = 2^{2n/3}x_L + 2^{n/3}x_M + x_R$
  
  - $y = 2^{2n/3}y_L + 2^{n/3}y_M + y_R$
Multiplying trinomials

\[(ax^2 + bx + c)(dx^2 + ex + f)\]
\[= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf\]

• How many multiplications?
Multiplying trinomials

\[(ax^2 + bx + c)(dx^2 + ex + f)\]

\[= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf\]

- 9 multiplications means 9 recursive calls
- Each multiplication is 1/3 the size of the original

\[T(n) = 9T\left(\frac{n}{3}\right) + O(n)\]
Multiplying trinomials

\[(ax^2 + bx + c)(dx^2 + ex + f)\]
\[= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf\]

- 9 multiplications means 9 recursive calls
- Each multiplication is 1/3 the size of the original

\[T(n) = 9T\left(\frac{n}{3}\right) + O(n)\]
Multiplying trinomials

\[(ax^2 + bx + c)(dx^2 + ex + f)\]
\[= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf\]

\[T(n) = 9T\left(\frac{n}{3}\right) + O(n)\]

• Master theorem

\[T(n) = aT\left(\frac{n}{b}\right) + O\left(n^d\right)\]
Master theorem applied to multiplying trinomials

• The recursion for the runtime of multiplying trinomials is

\[ T(n) = 9T\left(\frac{n}{3}\right) + O(n) \]

• So, we have that \( a = 9, \ b = 3, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in O\left(n^{\log_3 9}\right) = O(n^2) \]
Master theorem applied to multiplying trinomials

- The recursion for the runtime of multiplying trinomials is
  \[ T(n) = 9T\left(\frac{n}{3}\right) + O(n) \]

- So, we have that \( a = 9, b = 3, \) and \( d = 1. \) In this case, \( a > b^d \) so
  \[ T(n) \in O(n^{\log_3 9}) = O(n^2) \]
Multiplying trinomials

\[(ax^2 + bx + c)(dx^2 + ex + f)\]
\[= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf\]

- When multiplying binomials, we can reduce the number of multiplications from 4 to 3
- Similarly, when multiplying trinomials, we can reduce the number of multiplications from 9 to 5
- Then, the recursion becomes

\[T(n) = 5T\left(\frac{n}{3}\right) + O(n)\]
Multiplying trinomials

\[(ax^2 + bx + c)(dx^2 + ex + f)\]

\[= adx^4 + (ae + bd)x^3 + (af + be + cd)x^2 + (bf + ce)x + cf\]

• When multiplying binomials, we can reduce the number of multiplications from 4 to 3
• Similarly, when multiplying trinomials, we can reduce the number of multiplications from 9 to 5
• Then, the recursion becomes

\[T(n) = 5T\left(\frac{n}{3}\right) + O(n)\]
Master theorem applied to multiplying trinomials

- The recursion for the runtime of multiplying trinomials is
  \[ T(n) = 5T\left(\frac{n}{3}\right) + O(n) \]

- So, we have that \( a = 5, b = 3, \) and \( d = 1. \) In this case, \( a > b^d \) so
  \[ T(n) \in O\left(n^{\log_3 5}\right) = O(n^{1.43}) \]
Master theorem applied to multiplying trinomials

• The recursion for the runtime of multiplying trinomials is
\[ T(n) = 5T\left(\frac{n}{3}\right) + O(n) \]

• So, we have that \( a = 5, b = 3, \) and \( d = 1. \) In this case, \( a > b^d \) so
\[ T(n) \in O(n^{\log_3 5}) = O(n^{1.43}) \]
Dividing into $k$ subproblems

- What happens if we divide into $k$ subproblems each of size $n/k$?

$$\left( a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1x + a_0 \right) \left( b_{k-1}x^{k-1} + b_{k-2}x^{k-2} + \cdots + b_1x + b_0 \right)$$

- How many terms (multiplications)?
Dividing into $k$ subproblems

- What happens if we divide into $k$ subproblems each of size $n/k$?

\[ (a_{k-1}x^{k-1} + a_{k-2}x^{k-2} + \cdots + a_1x + a_0)(b_{k-1}x^{k-1} + b_{k-2}x^{k-2} + \cdots + b_1x + b_0) \]

- How many multiplications (terms)?
- There are $k^2$ multiplications. The recursion is

\[ T(n) = k^2T\left(\frac{n}{k}\right) + O(n) \]
Master theorem applied

- The recursion for the runtime is
  \[ T(n) = k^2 T \left( \frac{n}{k} \right) + O(n) \]
- So, we have that \( a = k^2, b = k, \) and \( d = 1. \) In this case, \( a > b^d \) so
  \[ T(n) \in O(n^{\log_b k^2}) = O(n^2) \]
Master theorem applied

• The recursion for the runtime is

\[ T(n) = k^2 T \left( \frac{n}{k} \right) + O(n^*) \]

• So, we have that \( a = k^2 \), \( b = k \), and \( d = 1 \). In this case, \( a > b^d \) so

\[ T(n) \in O(n^{\log_b k^2}) = O(n^2) \]
Cook-Toom algorithm

• In fact, if you split up your number into k equally sized parts, then you can combine them with $2k-1$ multiplications instead of the $k^2$ individual multiplications.

• This means that you can get an algorithm that runs in

$$T(n) = (2k - 1)T\left(\frac{n}{k}\right) + O(n)$$

3 instead of 4, $k = 2$, binomials
5 instead of 9, $k = 3$, trinomials
Master theorem applied

• The recursion for the runtime is

\[ T(n) = (2k - 1)T\left(\frac{n}{k}\right) + O(n) \]

• So, we have that \( a = 2k - 1, b = k, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in O\left(n^{\log_k(2k-1)}\right) = O\left(n^{\frac{\log(2k-1)}{\log k}}\right) \]

\[
T(n) = aT(n/b) + O(n^d) \\
\begin{cases} 
 0(n^d) & \text{if } a < b^d \\
 0(n^d \log n) & \text{if } a = b^d \\
 O(n^{\log_b a}) & \text{if } a > b^d 
\end{cases}
\]
Master theorem applied

• The recursion for the runtime is

\[ T(n) = (2k - 1)T\left(\frac{n}{k}\right) + O(n) \]

• So, we have that \( a = 2k - 1, \ b = k, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in O\left(n^{\log_k(2k-1)}\right) = O\left(n^{\log(2k-1)/\log k}\right) \]
Master theorem applied

• The recursion for the runtime is

\[ T(n) = (2k - 1)T \left( \frac{n}{k} \right) + O(n) \]

• So, we have that \( a = 2k - 1, b = k, \) and \( d = 1. \) In this case, \( a > b^d \) so

\[ T(n) \in O \left( n^{\log_k(2k-1)} \right) = O \left( n^{\frac{\log(2k-1)}{\log k}} \right) \]

\[ \log_2 3 = 1.58 \text{ binomials} \]
\[ \log_3 5 = 1.43 \text{ trinomials} \]
Cook-Toom algorithm

• \( T(n) = (2k - 1)T\left(\frac{n}{k}\right) + O(n) \)

• \( T(n) = O\left(n \frac{\log(2k-1)}{\log k}\right) \)

• We can have a near-linear time algorithm if we take \( k \) to be sufficiently large. The \( O(n) \) term in the recursion takes a lot of time the bigger \( k \) gets. So is it worth it to make \( k \) very large?
Next lecture

• Divide and conquer algorithms
  – Reading: Section 2.6