Chapter Nineteen: Cost Models
Up to this point, we have focused on computability without considering efficiency. As practical programmers, however, we are naturally interested in understanding the resource requirements of the code we write. How much time a program needs and how much memory space it needs are important practical questions. A cost model is a tool for answering such questions systematically. In this chapter, we define cost models for TMs and for our Java-like programs.
Outline

• 19.1 Asymptotic Notation
• 19.2 Properties of Asymptotics
• 19.3 Common Asymptotic Functions
• 19.4 A TM Cost Model
• 19.5 A Java Cost Model
Complexity

• Important questions about any program:
  – How much time does it need?
  – How much memory space does it need?
• A cost model is a tool for answering such questions systematically
• Asymptotic notation is a mathematical tool that we will use to construct our cost model
Big O

Let $f$ and $g$ be any two functions over $\mathbb{N}$. We say

$$f(n) \text{ is } O(g(n))$$

if and only if there exist natural numbers $c$ and $n_0$ so that for every $n \geq n_0$, $f(n) \leq c \, g(n)$.

**Intuition:** $f$ grows no faster than $g$. 

*Formal Language, chapter 19, slide 5*
If $f(n)$ is $O(g(n))$, then \( f(n) \) is an upper bound for \( g(n) \).
More About Upper Bounds

• When you assert that $f(n)$ is $O(g(n))$ you are saying that $f$ grows no faster than $g$
• That doesn't rule out the possibility that $f$ actually grows much slower than $g$
• For example, if $f(n)$ is $O(n^2)$, then these are also true (though less informative):
  – $f(n)$ is $O(n^3)$
  – $f(n)$ is $O(n^4)$
  – $f(n)$ is $O(n^{100})$
Big $\Omega$

Let $f$ and $g$ be any two functions over $\mathbb{N}$. We say

$$f(n) \text{ is } \Omega(g(n))$$

if and only if there exist natural numbers $c$ and $n_0$ so that for every $n \geq n_0$,

$$f(n) \geq \frac{1}{c} g(n)$$

Intuition: $f$ grows no slower than $g$. 

*Formal Language, chapter 19, slide 8*
Lower Bound

\[ f(n) \text{ is } \Omega(g(n)) \]

Formal Language, chapter 19, slide 9
More About Lower Bounds

• When you assert that $f(n)$ is $\Omega(g(n))$ you are saying that $f$ grows no slower than $g$
• That doesn't rule out the possibility that $f$ actually grows much faster than $g$
• For example, if $f(n)$ is $\Omega(n^3)$, then these are also true (though less informative):
  – $f(n)$ is $\Omega(n^2)$
  – $f(n)$ is $\Omega(n)$
  – $f(n)$ is $\Omega(1)$
Tight Bounds

• It is possible to have both at the same time:
  – $f(n)$ is $O(g(n))$
  – $f(n)$ is $\Omega(g(n))$

• There's a special notation for this important case...
Big $\Theta$

Let $f$ and $g$ be any two functions over $\mathbb{N}$. We say

$$f(n) \text{ is } \Theta(g(n))$$

if and only if $f(n)$ is $O(g(n))$ and $f(n)$ is $\Omega(g(n))$.

Intuition: $f$ grows at the same rate as $g$. 
Tight Bound

\[ f(n) \text{ is } \Theta(g(n)) \]

\[ c \cdot g(n) \]

\[ \frac{1}{c'} \cdot g(n) \]

*note: different constants for the \( \Theta \) and the \( \Omega \)*
Outline

• 19.1 Asymptotic Notation
• 19.2 Properties of Asymptotics
• 19.3 Common Asymptotic Functions
• 19.4 A TM Cost Model
• 19.5 A Java Cost Model
Simplifying Asymptotics

• Asymptotics have properties that allow many functions to be simplified
• That's what makes them so useful:
  – Describe a complicated function...
    • like $f(n) =$ the worst-case runtime of a given function on any input string of length $n$
  – ...by relating it to a simple function
    • by saying something like: $f(n) =$ $O(n^2)$
Eliminating Constant Factors

• Using asymptotics, we can *eliminate constant factors*
• For example, $O(3n^2)$ and $O(n^2)$ are equivalent
• Thus you should never rest with the conclusion that a function is $O(3n^2)$, because $O(n^2)$ is equivalent and simpler
Formally

Theorem 19.2.1, O version: for any nonzero $k \in \mathbb{N}$, $f(n)$ is $O(k \cdot g(n))$ if and only if $f(n)$ is $O(g(n))$.

• Proof:
  – Suppose $f(n)$ is $O(k \cdot g(n))$
  – So there exist positive constants $c$ and $n_0$ so that for every $n \geq n_0$, $f(n) \leq ck \cdot g(n)$
  – Let $c' = ck$. Now, for every $n \geq n_0$, $f(n) \leq c'\cdot g(n)$
  – Thus, by definition, $f(n)$ is $O(g(n))$
  – (The other direction of the proof is trivial: $g(n) \leq k \cdot g(n)$.)
Same For All Asymptotics

- The same thing applies to $\Omega$ and $\Theta$: drop constant factors
  - Theorem 19.2.1, $\Omega$ version: for any nonzero $k \in \mathbb{N}$, $f(n)$ is $\Omega(k \cdot g(n))$ if and only if $f(n)$ is $\Omega(g(n))$
  - Theorem 19.2.1, $\Theta$ version: for any nonzero $k \in \mathbb{N}$, $f(n)$ is $\Theta(k \cdot g(n))$ if and only if $f(n)$ is $\Theta(g(n))$
Eliminating Low-Order Terms

- Using asymptotics, we can eliminate all but the fastest growing from any sum of terms
- For example, $O(n^3 + n^2 + \log n)$ and $O(n^3)$ are equivalent
- Thus you should never rest with the conclusion that a function is $O(n^3 + n^2 + \log n)$, because $O(n^3)$ is equivalent and simpler
- If you ever find yourself writing "+" inside an asymptotic, think carefully!
Formally

Theorem 19.2.2, O version: let \{g_1, \ldots, g_m\} be any finite set of functions in which \(g_1\) is maximal, in the sense that every \(g_i(n)\) is \(O(g_1(n))\). Then for any function \(f\), \(f(n)\) is \(O(g_1(n) + g_2(n) + \ldots + g_m(n))\) if and only if \(f(n)\) is \(O(g_1(n))\).

• Proof:
  – Suppose \(f(n)\) is \(O(g_1(n) + g_2(n) + \ldots + g_m(n))\)
  – There exist natural numbers \(c_0\) and \(n_0\) so that for every \(n \geq n_0\), \(f(n) \leq c_0 \cdot ((g_1(n) + g_2(n) + \ldots + g_m(n))\)
  – We are also given that every term \(g_i(n)\) is \(O(g_1(n))\)
  – There exist natural numbers \(\{c_1, \ldots, c_m\}\) and \(\{n_1, \ldots, n_m\}\) such that, for every \(i\) and every \(n \geq \max(n_0, \ldots, n_m)\), \(g_i(n) \leq c_i g_1(n)\)
Proof, continued:

- Therefore, for every \( n \geq \max(n_0, \ldots, n_m) \),
  \[
  f(n) \leq c_0 \cdot ((g_1(n) + g_2(n) + \ldots + g_m(n))
  \leq c_0 \cdot ((c_1g_1(n) + c_2g_1(n) + \ldots + c_mg_1(n))
  = c_0 \cdot (c_1 + \ldots + c_m) \cdot g_1(n)
  
  - By choosing \( c' = c_0(c_1 + \ldots + c_m) \) and \( n' = \max(n_0, \ldots, n_m) \), we can restate this as follows: for every \( n \geq n' \), \( f(n) \leq c' \cdot g_1(n) \)
  
  - Thus, by definition, \( f(n) \) is \( O(g_1(n)) \)
  
  - (Proof of the other direction is trivial, because \( g_1(n) \leq g_1(n) + g_2(n) + \ldots + g_m(n) \))
Same For All Asymptotics

- The same thing applies to $\Omega$ and $\Theta$: eliminate all but the fastest-growing terms from any sum
- Always with the condition that all $g_i(n)$ are $O(g_1(n))$:
  - Theorem 19.2.2, $\Omega$ version:
    $f(n)$ is $\Omega(g_1(n) + g_2(n) + \ldots + g_m(n))$
    if and only if $f(n)$ is $\Omega(g_1(n))$
  - Theorem 19.2.2, $\Theta$ version:
    $f(n)$ is $\Theta(g_1(n) + g_2(n) + \ldots + g_m(n))$
    if and only if $f(n)$ is $\Theta(g_1(n))$
Polynomials In Particular

• Apply our two simplifications:
  – Choose the fastest-growing term (the highest-order term)
  – Drop the constant factors (all coefficients)
• For example:
  – \( O(4n^2 + 4n + 2) \) simplifies to \( O(n^2) \)
Asymptotic Products

• Theorem 19.2.3, $O$ version:
  
  If $f_1(n)$ is $O(g_1(n))$,  
  and $f_2(n)$ is $O(g_2(n))$,  
  then $f_1(n) \cdot f_2(n)$ is $O(g_1(n) \cdot g_2(n))$

• Ditto for $\Omega$ and $\Theta$

• Useful for analyzing loops:
  – If each iteration takes $O(g_1(n))$ time...
  – ...and there are $O(g_2(n))$ iterations...
  – ...we can conclude the total time for the loop is $O(g_1(n) \cdot g_2(n))$
Symmetry

• Theorem 19.2.4: for any functions $f$ and $g$ over $\mathbb{N}$, $f(n)$ is $O(g(n))$ if and only if $g(n)$ is $\Omega(f(n))$
• Proof follows directly from the definitions
• Interesting but not often useful
• We use asymptotics to characterize complicated functions by relating them to simple functions
• Reversing the direction is rarely helpful
Outline

• 19.1 Asymptotic Notation
• 19.2 Properties of Asymptotics
• 19.3 Common Asymptotic Functions
• 19.4 A TM Cost Model
• 19.5 A Java Cost Model
Common Asymptotic Functions

- Start with functions typically encountered when analyzing the resource requirements of computer programs
- Apply the simplifications just described
- You end up with a kind of standard vocabulary of functions
- These occur so often in asymptotic analysis that they have special names
Our Standard Functions

- $\Theta(1)$: constant
- $\Theta(\log n)$: logarithmic
- $\Theta(n)$: linear
- $\Theta(n \log n)$: loglinear
- $\Theta(n^k)$: polynomial
  - $k = 2$: quadratic
  - $k = 3$: cubic
- $\Theta(2^n)$: exponential

Shown for tight bounds; the same names are used with lower and upper bounds
Asymptotic Constant

• $f(n)$ is $\Theta(1)$
  – That is, $f(n)$ is $\Theta(g(n))$ where $g(n) = 1$
  – This $g(n)$ does not grow as $n$ grows

• Note that $\Theta(1)$ is the same as $\Theta(k)$ for any positive constant $k$

• But simpler is better, so we always write $\Theta(1)$ instead of any other constant
Logarithms

• $x = \log_b n$ if and only if $b^x = n$

• Here, we often want integer logs:

$$x = \left\lfloor \log_b n \right\rfloor$$

• Intuition: the number of times you can divide $n$ by $b$ until you get a number that is $\leq 1$

• In computer science, logs are usually base 2
Asymptotic Logarithmic

• Different contexts, different bases:
  – In CS, often $\log_2 n$
  – In calculus, often $\log_e n$
  – In calculators, often $\log_{10} n$

• But for any two constants $a$ and $b$, $\log_a n$ and $\log_b n$
differ only by a constant factor
  – $\log_a n = (\log_a b) \log_b n$

• Constant factors are dropped in asymptotics
• So the convention is to omit the base and simply write
  $\Theta(\log n)$
Practical Note

- $\log n$ is a very slow-growing function
- For many practical purposes, $\log n$ is effectively bounded by a constant
- For example, for any positive number $n$ representable as a Java int, $\log_2 n < 32$
Asymptotic Linear

• $f(n)$ is $\Theta(n)$
  – That is, $f(n)$ is $\Theta(g(n))$ where $g(n) = n$
• As always, constant factors are dropped
• For example:
  – you would never write $\Theta(3n)$
  – $\Theta(n)$ is equivalent and simpler
Asymptotic Loglinear

- \( f(n) \) is \( \Theta(n \log n) \)
- The product of \( n \) times \( \log n \) arises quite often in the analysis of algorithms
- It grows just slightly faster than \( n \), but not nearly as fast as \( n^2 \)
Asymptotic Polynomial

- $f(n)$ is $\Theta(n^k)$ for some constant $k$
  - Already seen $\Theta(n^0)$: constant
  - Already seen $\Theta(n^1)$: linear
  - $\Theta(n^2)$: quadratic
  - $\Theta(n^3)$: cubic
- As always, constant factors and slower-growing terms are dropped
- For example:
  - You would never write $\Theta(3n^2 + 4n + 2)$
  - $\Theta(n^2)$ is equivalent and simpler
Sums

- Two sums that occur often in analysis of algorithms

\[ f_1(n) = \sum_{i=1}^{n} n = n + n + \cdots + n = n \cdot n = n^2 \]

\[ f_2(n) = \sum_{i=1}^{n} i = 1 + 2 + \cdots + n = \frac{n(n+1)}{2} \]

- Note these are the same asymptotically
- They are both \( \Theta(n^2) \)
Asymptotic Exponential

- $f(n)$ is $\Theta(b^n)$ for some constant $b > 1$
- Most commonly, $\Theta(2^n)$
- Note that (unlike logarithmics) the base is significant: $\Theta(2^n)$ is not the same as $\Theta(3^n)$
Rates of Growth

• Slowest to fastest:
  – $\Theta(1)$: constant
  – $\Theta(\log n)$: logarithmic
  – $\Theta(n)$: linear
  – $\Theta(n \log n)$: loglinear
  – $\Theta(n^k)$: polynomial, (for $k > 1$)
  – $\Theta(2^n)$: exponential
Outline

• 19.1 Asymptotic Notation
• 19.2 Properties of Asymptotics
• 19.3 Common Asymptotic Functions
• 19.4 A TM Cost Model
• 19.5 A Java Cost Model
Time And Space For TMs

• We define the time and space required by a TM $M$ on input $x \in \Sigma^*$ as follows:
  – $time(M, x) =$ the number of moves made by $M$ on input $x$
  – $space(M, x) =$ the length of tape visited by $M$ on input $x$

• Worst-case cost functions remove specific strings from the picture:
  – $worst-case-time(M, n) =$ max $time(M, x)$ with $|x| = n$
  – $worst-case-space(M, n) =$ max $space(M, x)$ with $|x| = n$

• Can also consider best-case and average cases, variously defined
• Worst-case complexity is what we'll focus on
Example: Linear Time And Space

\[ L(M) = L(a^*b^*c^*) \]

- For \( x \in L(M) \)
  - Always moves right until \( B \) following input
  - So \( \text{time}(M, x) = |x| + 1 \)
  - And \( \text{space}(M, x) = |x| + 2 \) (counting final position)
- For \( x \notin L(M) \), stops earlier
- So worst-case time and space are linear:
  - \( \text{worst-case-time}(M, n) \) is \( \Theta(n) \)
  - \( \text{worst-case-space}(M, n) \) is \( \Theta(n) \)
Example: Quadratic Time, Linear Space

$L(M) = \{a^n b^n\}$

- Never writes non-$B$ over $B$; so the number of non-$B$ symbols is never greater than the original length $m$
- When any state is entered, $\leq m$ moves before exit
- On every 1, 2, 3, 4 cycle, erases one $a$ and one $b$; so no more than $m/2$ cycles before all erased
- $O(m)$ cycles back and forth, $O(m)$ steps each:
  - worst-case-time($M, m$) is $O(m^2)$
  - worst-case-space($M, m$) is $O(m)$
Example Continued

\[ L(M) = \{a^n b^n\} \]

- We can also show that the bounds are tight:
  - worst-case-time\((M, m)\) is \(\Theta(m^2)\)
  - worst-case-space\((M, m)\) is \(\Theta(m)\)
- You can even construct formulas for worst-case time and space without using asymptotics, but it's tricky
- Asymptotics are very useful for cost analysis of TMs
- They're indispensable when TMs are described but not actually constructed

*Formal Language*, chapter 19, slide 43
Outline

• 19.1 Asymptotic Notation
• 19.2 Properties of Asymptotics
• 19.3 Common Asymptotic Functions
• 19.4 A TM Cost Model
• 19.5 A Java Cost Model
Asymptotic Analysis

- We won't model exactly how much time and space each primitive operation takes ("a+b takes 14 ns")
- Instead, we'll state assumptions asymptotically ("a+b takes \( \Theta(1) \) time")
- So our conclusions will apply not just to a particular experimental setup, but to any implementation that meets our asymptotic assumptions
Key Issue: \texttt{int} Values

- We've been assuming unlimited \texttt{int}
- So it wouldn't be fair now to assume \texttt{int} operations can be done in constant time
- Instead, we'll model an implementation of \texttt{int} that is a bit like \texttt{String}:
  - A reference to an immutable block of memory
    - So parameter-passing and assignment take constant time (because they copy the reference, not the data)
  - Allocated whenever a new value is constructed
    - So modifying an \texttt{int}, as in \texttt{x=x+1}, takes time proportional to the memory space taken up by the \texttt{int}
Basic Java Cost Model

- All values take space proportional to the length of their representation as Java constants.
  - E.g., $\Theta(\log n)$ space for integer value $n$
- The time required for any operator is $\Theta(n)$, where $n$ is the total size of the data read and written
  - E.g., $\Theta(\log a)$ time to compute $a+1$
- $\Theta(1)$ time and space overhead for method calls
  - Parameter passing, method dispatch, initial stack allocation, returning a value
- Predefined methods:
  - `s.length()` takes $\Theta(1)$ time and space
  - `s.charAt(i)` takes $\Theta(\log i)$ time and $\Theta(1)$ space
  - We'll state other assumptions as needed
Example

boolean ax(String p) {
    return (p.length() != 0 && p.charAt(0) == 'a');
}

• A decision method for \{ax | x \in \Sigma^*\}
• Let $n$ be the input size: $n = |p|$
• We'll analyze to find worst-case time as a function of $n$
1. boolean ax(String p) {
   return (1)
   p.length(); (1)
   != (1)
   0 (1)
   && (1)
   p.charAt(0) (1)
   == (1)
   'a'); (1)
10. }

- Time complexity does not depend on $n$
  - (We're assuming the $!=$ operator on line 4 does not need to look at the whole number to zero-test)
  - The sum is the fastest-growing asymptotic in the column: $\text{worst-case-time}(ax, n)$ is $\Theta(1)$
Space complexity also does not depend on $n$

- The space taken by $p$ is $\Theta(n)$, of course
- But in our model, $p$ is passed by reference, not copied into a new block of memory
- Ditto for $p.length()$ on line 3: the value takes $\Theta(\log n)$ space, but we don't recopy that value

So worst-case-space($ax$, $n$) is $\Theta(1)$
Constant Terms

• Constant space and time are already given as the overhead of calling a method in the first place
• From now on, we'll simplify the space analysis by ignoring constant-space needs
• Ditto for time analysis, for constant-time operations repeated for any constant number of iterations
Example

```java
boolean anbn(String x) {
    if (x.length() % 2 != 0) return false;
    for (int i = 0; i < x.length() / 2; i++)
        if (x.charAt(i) != 'a') return false;
    for (int i = x.length() / 2; i < x.length(); i++)
        if (x.charAt(i) != 'b') return false;
    return true;
}
```

- A decision method for \( \{a^n b^n\} \)
- Let \( n \) be the input size: \( n = |x| \)
- We'll analyze to find worst-case time as a function of \( n \)
- First, let's get a lower bound on it...
1. boolean anbn(String x) {
2.     if (x.length() % 2 != 0) \(\Omega(\log n)\) 1 \(\Omega(\log n)\)
3.         return false;
4.     for (int i = 0;
5.         i < x.length() / 2; \(\Omega(\log n)\) \(\Omega(n)\) \(\Omega(n \log n)\)
6.         i++) \(\Omega(1)\) \(\Omega(n)\) \(\Omega(n)\)
7.         if (x.charAt(i) != 'a') \(\Omega(1)\) \(\Omega(n)\) \(\Omega(n)\)
8.         return false;
9.     for (int i = x.length() / 2;
10.        i < x.length(); \(\Omega(\log n)\) \(\Omega(n)\) \(\Omega(n \log n)\)
11.        i++) \(\Omega(\log n)\) \(\Omega(n)\) \(\Omega(n \log n)\)
12.        if (x.charAt(i) != 'b') \(\Omega(\log n)\) \(\Omega(n)\) \(\Omega(n \log n)\)
13.        return false;
14.     return true;
15. }

\[\Omega(n \log n)\] + 

\textit{i} can be as low as 0 here

\textit{i} is at least \(\frac{n}{2}\) here

- The total column shows the minimum cost for any iteration of the line, times the minimum number of iterations of that line

\textit{Formal Language}, chapter 19, slide 53
This uses the fact that $i \leq n$ everywhere

• Lower bound and upper bound match, so $\text{worst-case-time}(\text{anbn}, n)$ is $\Theta(n \log n)$
Extra Log Factor

- worst-case-time(anbn, n) was clearly going to be Ω(n), because we have to look at all n symbols in the worst case
- But where did the extra log factor come from?
- Unavoidable in our cost model: accessing the ith character of a string takes time proportional to log i
- Next, space analysis...
• Ignoring constant-space needs
• Ignoring iteration count: space used on one iteration can be reused on the next
• $\text{worst-case-space}(\text{anbn}, n)$ is $\Theta(\log n)$
Example

String matchingAS(String s) {
    String r = "";
    for (int i = 0; i < s.length(); i++)
        r += 'a';
    return r;
}

• (Not a decision method, of course)
• Let $n$ be the input size: $n = |s|$
• We'll analyze to find worst-case time as a function of $n$
• First, let's get a lower bound on it...
1. String matchingAs(String s) {
2.     String r = "";
3.     for (int i = 0;
4.         i < s.length();
5.         i++)
6.         r += 'a';
7.     return r;
8. }

- Rather weak lower bounds
  - i can be as small as 0
  - r can be as small as ""

Formal Language, chapter 19, slide 58
• \(\Omega(n \log n)\) and \(O(n^2)\)
• Usually, we want to refine our analysis until the upper and lower bounds meet
• In this case, we knew that the lower-bound analysis was weak
• Look again at line 6...
1. String matchingAs(String s) {
2.     String r = "";
3.     for (int i = 0;
4.         i < s.length();
5.         i++)
6.         r += 'a';
7.     return r;
8. }

• Total time on line 6 is more than just the minimum time for any iteration, times the number of iterations
• |r| starts at 0 but grows by 1 on each iteration
• So the total on line 6 is $g(n) = \sum_{k=1}^{n} t(k)$ where $t(k)$ is $\Omega(k)$
• As we've seen, this $g(n)$ is $\Omega(n^2)$
• That matches our upper bound, so $\text{worst-case-time(matchingAs, n)}$ is $\Theta(n^2)$

Formal Language, chapter 19, slide 60
Largest value constructed at line 6 has the same length as the input string

\textit{worst-case-space} (\texttt{matchingAs}, n) is $\Theta(n)$
Final Example

boolean isPrime(int x) {
    if (x < 2) return false;
    for (int i = 2; i < x; i++)
        if (x % i == 0) return false;
    return true;
}

• Decision method for the set of prime numbers
• Analysis in terms of input size $n$
• That's the amount of space for $x$
• For some constant base $b$, $x$ is $\Theta(b^n)$
• So we'll get this upper bound for time...
• Lower bound would be weaker on line 4, but would give the same total, $\Omega(nb^n)$
• So *worst-case-time*(isPrime, $n$) is $\Theta(nb^n)$
• In terms of the value of $x$, that's $\Theta(x \log x)$
• But we always treat complexity as a function of the size of the input string, not the magnitude of the number that string represents

```java
1. boolean isPrime(int x) {
2.     if (x < 2) return false;  // O(n) O(1) O(n)
3.     for (int i = 2; i < x; i++)  // O(n) O(b^n) O(nb^n)
4.         if (x % i == 0)  // O(1) O(b^n) O(b^n)
5.             return false;  // O(1) O(b^n) O(b^n)
6.     }  // return true;  // +_________
7.     return true;  // O(n b^n)
8. }
```

*Formal Language*, chapter 19, slide 63
1. boolean isPrime(int x) {
    2.     if (x < 2) return false; // $O(n)$
    3.     for (int i = 2;
    4.         i < x;
    5.         i++) // $\Theta(n)$
    6.         if (x % i == 0) // $\Theta(n)$
    7.         return false;
    8.     return true;
    9. } // $+\infty$
   10. return true;
   11. } \[ \in \leq x \]
   12. \[ \in \leq \left\lceil \frac{x}{2} \right\rceil - 1 \]

- worst-case-space(isPrime, n) is $\Theta(n)$

*Formal Language*, chapter 19, slide 64