Performance and Asymptotics

		(Jordan time.)	
	Miles Jones	**MTThF 8:30-9:50~~pm~~** am	**CSE 4258**

August 4, 2016

General questions to ask about algorithms

1) **What** problem are we solving? SPECIFICATION

2) **How** do we solve the problem? ALGORITHM DESCRIPTION

3) **Why** do these steps solve the problem? CORRECTNESSS

4) **When** do we get an answer? RUNNING TIME PERFORMANCE

Counting operations: WHEN

Measure …

Comparisons of list elements!

~~Time~~

Number of operations

For selection sort (MinSort), how many times do we have to compare the values of some pair of list elements?

What other operations does MinSort do?

Counting operations: WHEN

What kinds of operations constitute a single step?

- Comparisons
- assignment
- swapping 2 entries.
- (simple) addition.

Counting operations: WHEN

Number of operations

- Boolean operation (and, or, not, etc.)
  - Example: (If (X=Y) AND (X=Z) then…)
- Increment a counter.
  - Example: (i++)
- Arithmetic Operations. (+, -, *, etc…) (simple.)
  - Example: x:=y+z
- Comparison (which is larger? Are they equal?)
  - Example If (x>y) then
- Access a position in an array.
  - Example: x:=A[i]

Counting operations: WHEN

- We have listed operations that may be considered as single steps. But we’ve seen that whether they are really sinle steps or unravel into mini-algorithms of their own depends on circumstances.

- “single step” may be ambiguous for example, multiplication and addition are “single steps” if the size of the input is relatively small.

Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

every passage through the inner loop takes $\leq t_1 + t_2$ time.

```
procedure selection sort(a_1, a_2, ..., a_n: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( a_j < a_m ) then m := j
    interchange a_i and a_m

{ a_1, ..., a_n  is in increasing order}
```

comparison t_1

assignment t_2

$(n-i)\cdot(t_1+t_2)$

For each value of i, compare

(n-i)

pairs of elements.

$(n-1) + (n-2) + \ldots + (1)$

$= n(n-1)/2 \, (t_1+t_2)$

Sum of positive integers up to (n-1)

it goes through inner loop n-i times

there are $\leq 2\left(\frac{n(n-1)}{2}\right)$ operations.

Counting operations

When do we get an answer? RUNNING TIME PERFORMANCE

↕

Counting number of times list elements are compared

operations.

Runtime performance

Algorithm: problem solving strategy as a sequence of steps

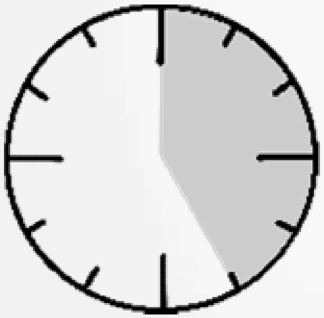
Examples of steps

- Comparing list elements (which is larger?)
- Accessing a position in a list (probe for value)
- Arithmetic operation (+, -, *,…)
- etc.

"Single step" depends on context

Runtime performance

How long does a "single step" take?



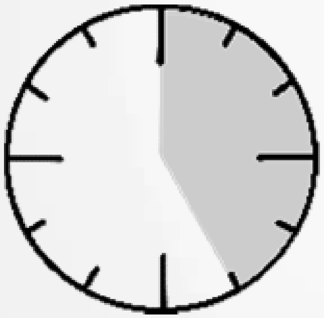
Some factors

- Hardware CPU, RAM, cache, temp.
- Software compiler, programming language, types of multiplication

Discuss & list the factors that could impact how long a single step takes

Runtime performance

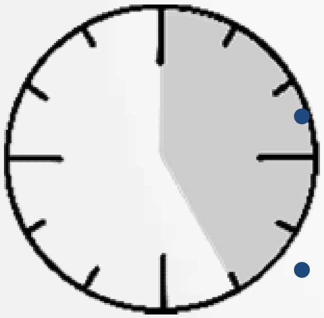
How long does a "single step" take?



Some factors

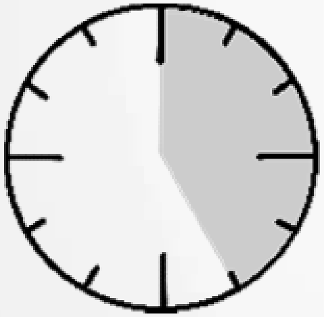
- Hardware (CPU, climate, cache …)
- Software (programming language, compiler)

Runtime performance



- Most instructions are carried out in the CPU. The clock sets the rate at which the CPU carries out instructions. For a first pass, this determines processing speed.
- Different processing units are optimized for different types of instructions. For example, graphical processing units (GPU) are optimized for floating point arithmetic.
- Processors generate a lot of heat which can slow down your computer.
- Cache memory is much faster than RAM which is faster than disk. The time to read data from memory depends on where it is stored. So having quick-access can speed up performance.

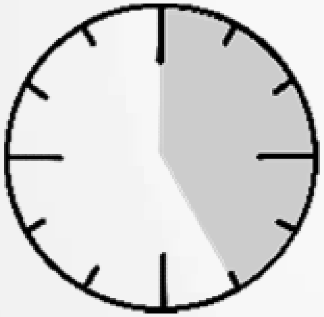
Runtime performance



- What we count as a step depends on the scale and circumstances of our problem.
- Different types of steps require different exact times.
- The algorithm designer controls how many times steps are performed, but the exact time steps take is outside the control of the designer.

Runtime performance

The time our program takes will depend on



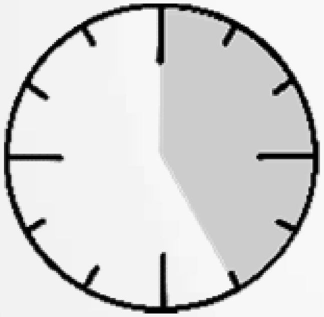
Input size

Number of steps the algorithm requires

Time for each of these steps on our system

Runtime performance

It is impossible to give exact amounts of time an algorithm takes. We will estimate the time based on the input size



Best-case time

Worst-case time

Average-case time

Runtime performance

TritonSort is a project here at UCSD that has the world record sorting speeds, 4 TB/minute. It combines algorithms (fast versions of radix sort and quicksort), parallelism (a tuned version of Hadoop) and architecture (making good use of memory hierarchy by minimizing disc reads and pipelining data to make sure that processors always have something to compare). I think it is a good example of the different hardware, software and algorithm components that affect overall time. This is a press release

CNS Graduate Student Once Again Breaks World Record! (2014)Michael Conley, a PhD student in the CSE department, once again won a data sort world record in multiple categories while competing in the annual Sort Benchmark competition. Leading a team that included Professor George Porter and Dr. Amin Vahdat, Conley employed a sorting system called Tritonsort that was designed not only to achieve record breaking speed but also to maximize system resource utilization. Tritonsort tied for the "Daytona Graysort" category and won outright in both the "Daytona" and "Indy" categories of the new "Cloudsort" competition. To underscore the effectiveness of their system resource utilization scheme as compared to the far more resource intensive methods followed by their competitors, it's interesting to note that the 2011 iteration of Tritonsort still holds the world record for the "Daytona" and "Indy" categories of the "Joulesort" competition.

Runtime performance

Ignore what we can't control

Goal:

Estimate time as a function of the size of the input, n

Focus on how time scales for large inputs

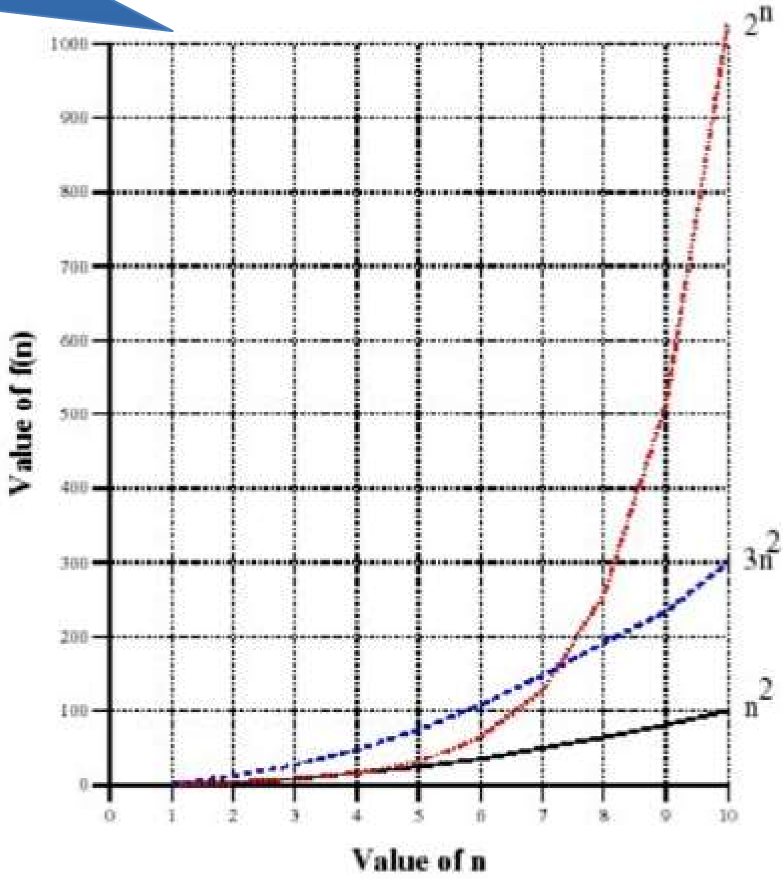
Rate of growth

Ignore what we can't control

Focus on how time scales for large inputs

$3n^2 \sim n^2$

$2^n > n^2$



Which of these functions do you think has the "same" rate of growth?

A. All of them
B. 2^n and n^2
C. n^2 and 3n^2
D. They're all different

Definition of Big O

Ignore what we can't control

Focus on how time scales for large inputs

For functions $f(n) : \mathbb{N} \to \mathbb{R}, g(n) : \mathbb{N} \to \mathbb{R}$ we say

$3n^2 \in O(n^2)$

let $c = 5$

$3n^2 \leq 5n^2$

$$\boxed{f(n) \in O(g(n))}$$

to mean there are constants, C and k such that $|f(n)| \leq C|g(n)|$ for all n > k.

$n^2 \in O(2^n)$

$n^2 \in O(3n^2)$

Rosen p. 205

Definition of Big O

Ignore what we can't control

Focus on how time scales for large inputs

For functions $f(n): \mathbb{N} \rightarrow \mathbb{R}, g(n): \mathbb{N} \rightarrow \mathbb{R}$ we say

$$\boxed{f(n) \in O(g(n))}$$

to mean there are constants, C and k such that $|f(n)| \leq C|g(n)|$ for all n > k.

Rosen p. 205

Definition of Big O

For functions $f(n) : \mathbb{N} \to \mathbb{R}, g(n) : \mathbb{N} \to \mathbb{R}$ we say

$3n^2 + 2n \in O(n^2)$

$$f(n) \in O(g(n))$$

to mean there are constants, C and k such that $|f(n)| \leq \underline{C}|g(n)|$ for all n > k.

~~$3n^2 + 2n > \frac{1}{3}n^2$~~

Example: $f(n) = 3n^2 + \cancel{2n}$ $g(n) = n^2$

$3n^2 + 2n \leq 5n^2$

What constants can we use to prove that $f(n) \in O(g(n))$

A. C = 1/3, k = 2

[✓ B. C = 5, k = 1]

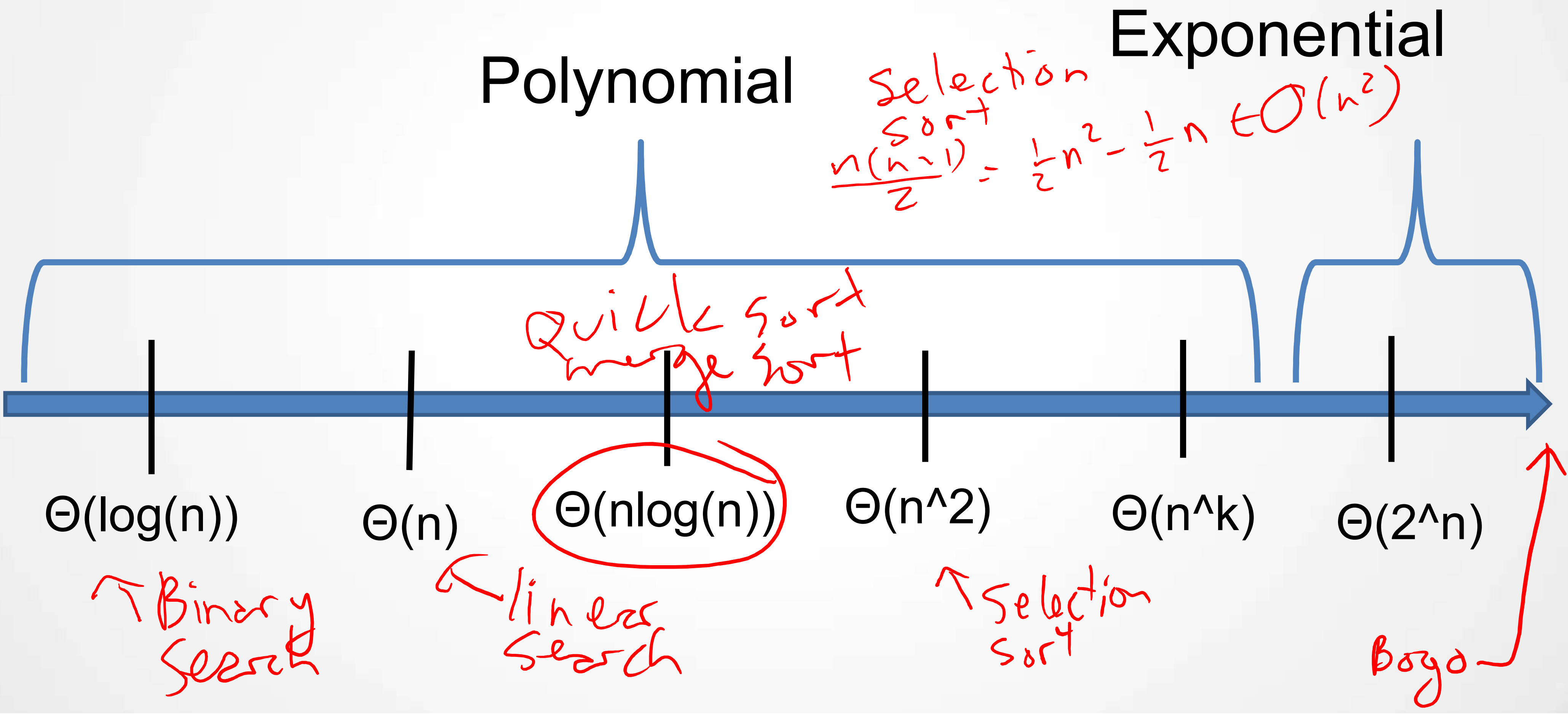
[✓ C. C = 10, k = 2]

D. None: f(n) isn't big O of g(n).

$3n^2 + 2n < 10n^2$

for all $n > 2$

Definition of Big O



Polynomial

Exponential

Selection sort

$$\frac{n(n-1)}{2} = \frac{1}{2}n^2 - \frac{1}{2}n \in \Theta(n^2)$$

Quick sort

merge sort

Θ(log(n)) Θ(n) Θ(nlog(n)) Θ(n^2) Θ(n^k) Θ(2^n)

Binary Search

linear search

Selection sort

Bogo

Big O : Notation and terminology

"f(n) is big O of g(n)"

A family of functions which grow no faster than g(n)

$$f(n) \in O(g(n))$$

What functions are in the family O(n^2) ?

$n^2, n, \sqrt{n}, \log n, n \log n, \sum_{i=1}^{n} i,$

$1, \log \log n$

Big O : Potential pitfalls

"f(n) is big O of g(n)"

$3n^2 \in O(n^2)$

$$f(n) \in O(g(n))$$

- The **value** of f(n) might always be bigger than the **value** of g(n).
- O(g(n)) contains functions that grow **strictly slower** than g(n).

false

$o(g(n))$

Big O : How to compute?

Is f(n) big O of g(n) ? i.e. is $f(n) \in O(g(n))$?

$\cancel{3}n^7 + \cancel{2n^6}$

Approach 1: Look for constants C and k.

Approach 2: Use properties

Domination *If f(n) <= g(n) for all n then f(n) is big-O of g(n).*

Transitivity *If f(n) is big-O of g(n), and g(n) is big-O of h(n), then f(n) is big-O of h(n)*

Additivity/ Multiplicativity *If f(n) is big-O of g(n), and if h(n) is nonnegative, then f(n) * h(n) is big-O of g(n) * h(n) … where * is either addition or multiplication*

Sum is maximum *f(n)+g(n) is big-O of the max(f(n), g(n))*

Ignoring constants *For any constant c, cf(n) is big-O of f(n)*

Rosen p. 210-213

Big O : How to compute?

Is f(n) big O of g(n) ? i.e. is $f(n) \in O(g(n))$?

Approach 1: Look for constants C and k.

Approach 2: Use properties

Domination *If f(n) <= g(n) fo n then f(g-O of g(n)*
Transitivity *is big-O g-O (n), then f(n) is*
big-O

Additivity/ *is nonnegative,*
*(n) … where * is*
tion.

Sum is maximu *f(g(n) is big-O of the max(f(n), g(n))*
Ignoring consta ts *for any constant c, cf(n) is big-O of f(n)*

Look at terms one-by-one and drop constants. Then only keep maximum.

Rosen p. 210-213

Big O : How to compute?

Is f(n) big O of g(n) ? i.e. is $f(n) \in O(g(n))$?

$f(n) \in O(g(n))$

$|f(n)| \le c\,|g(n)|$

Approach 3. The limit method. Consider the limit

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \; .$$

$f \in O(g)$

$\frac{f(n)}{g(n)} \le c$

I. If this limit exists and is 0: then f(n) grows strictly slower than g(n).

$f \in O(g)$

II. If this limit exists and is a constant c > 0: then f(n), g(n), grow at the same rate.

III. If the limit tends to infinity: then f(n) grows strictly faster than g(n). $f \notin O(g)$

IV. if the limit doesn't exist for a different reason … use another approach!

$f \in \Theta(g)$

Other asymptotic classes

Rosen p. 214-215

$f(n) \in O(g(n))$ $[\leq]$

means there are constants, C and k such that $|f(n)| \leq C|g(n)|$ for all n > k.

$f(n) \in \underline{\Omega}(g(n))$

means $g(n) \in O(f(n))$ $[\geq]$

$f(n) \in \Theta(g(n))$

means $f(n) \in O(g(n))$ and $g(n) \in O(f(n))$ $[=]$

$f(n) \in \Omega(g(n))$

What functions are in the family $\Theta(n^2)$? $3n^2, 5n^2+2n, \sum_{i=1}^{n} i$

Selection Sort (MinSort) Performance

Rosen page 210, example 5

Number of comparisons of list elements

$(n-1) + (n-2) + \ldots + (1)$

Sum of positive integers up to (n-1)

Rewrite this formula in order notation:

~~A. O(n)~~
[✓ B. O(n(n-1))]
[✓ C. O(n^2)]
~~D. O(1/2)~~
E. None of the above

Selection Sort (MinSort) Performance

Rosen page 210, example 5

Number of comparisons of list elements

$$(n-1) + (n-2) + \ldots + (1) = \boxed{n(n-1)/2} \in O(n^2)$$

Sum of positive integers up to (n-1)

Rewrite this formula in order notation:

A. O(n)
B. O(n(n-1))
C. $O(n^2)$
D. O(1/2)
E. None of the above

Selection Sort (MinSort) Pseudocode

Rosen page 203, exercises 41-42

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j$ < $a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$  is in increasing order}
```

Computing the big-O class of algorithms

How to deal with …

Basic operations

Consecutive (non-nested) code

Loops (simple and nested)

Subroutines

Computing the big-O class of algorithms

How to deal with …

Basic operations : operation whose time doesn't depend on input

Consecutive (non-nested) code : one operation followed by another

add

Loops (simple and nested) : while loops, for loops

multiply
(body · # of iterations)

Subroutines : method calls

Computing the big-O class of algorithms

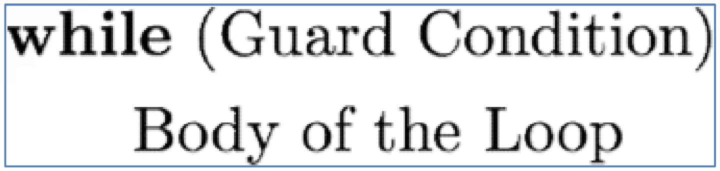
Consecutive (non-nested) code : Run Prog_1 followed by Prog_2

If Prog_1 takes O(f(n)) time and Prog_2 takes O(g(n)) time, what's the big-O class of runtime for running them consecutively?

A. O(f(n) + g(n)) [[sum]]
B. O(f(n) g(n)) [[multiplication]]
C. O(g(f(n))) [[function composition]]
D. O(max (f(n), g(n)))
E. None of the above.

Computing the big-O class of algorithms

Simple loops:



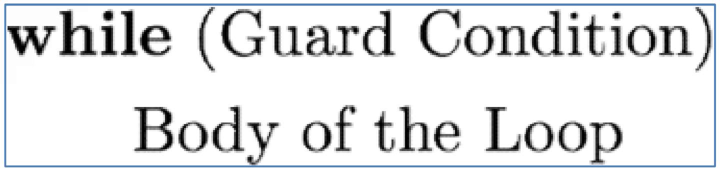
```
while (Guard Condition)
    Body of the Loop
```

What's the runtime?

$O(\text{time}(\text{Body})) \cdot$ # of iterations.

Computing the big-O class of algorithms

Simple loops:



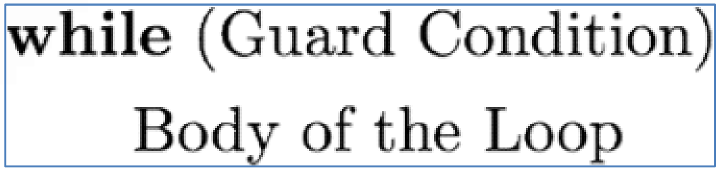
```
while (Guard Condition)
    Body of the Loop
```

What's the runtime?

Number of iterations *times* the time it takes for the body of the loop.

Computing the big-O class of algorithms

Simple loops:

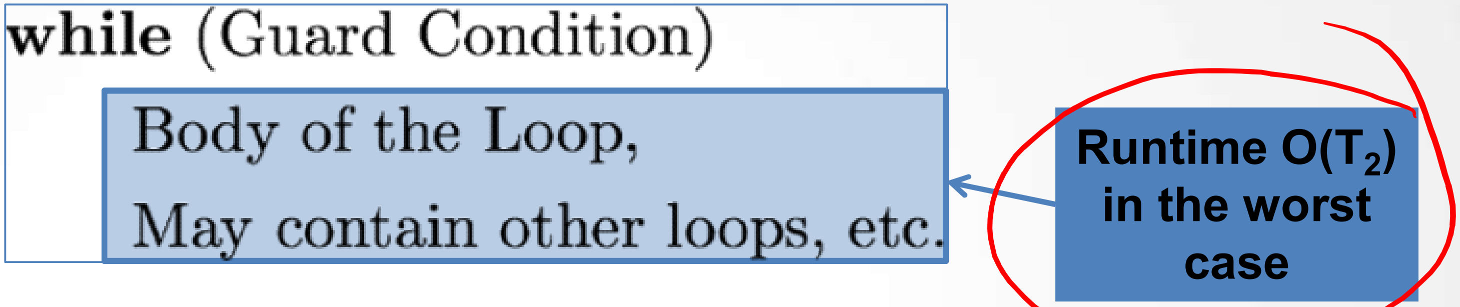


```
while (Guard Condition)
    Body of the Loop
```

If Guard Condition uses basic operations and body of the loop is constant time, then runtime is of the same order as the number of iterations.

Computing the big-O class of algorithms

Nested code:



```
while (Guard Condition)
    Body of the Loop,
    May contain other loops, etc.
```

Runtime $O(T_2)$ in the worst case

If Guard Condition uses basic operations and body of the loop has ~~constant time~~ runtime $O(T_2)$ in the worst case, then runtime is

$$O(T_1 T_2)$$

where T_1 is the bound on the number of iterations through the loop.

Product rule

Computing the big-O class of algorithms

Subroutine Call method S on (some part of) the input.

If sub-routine S has runtime $T_S(n)$ and we call S at most T_1 times,

A. Total time for all uses of S is $T_1+T_S(n)$
B. Total time for all uses of S is $\max(T_1,T_S(n))$
C. Total time for all uses of S is $T_1T_S(n)$
D. None of the above

loop T_1 iterations

Body $T_s(n)$

Computing the big-O class of algorithms

Subroutine Call method S on (some part of) the input.

If sub-routine S has runtime is O ($T_S(n)$) and
if we call S at most T_1 times, then runtime is

$$O(\ T_1 T_S(m)\ )$$

where m is the size of biggest input given to S.

Distinguish between the size of input to subroutine, m, and the size of the original input, n, to main procedure!

Selection Sort (MinSort) Pseudocode

Before, we counted comparisons, and then went to big-O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j < a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$  is in increasing order}
```

Selection Sort (MinSort) Pseudocode

Before, we counted comparisons, and then went to big-O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j$ < $a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$ is in increasing order}
```

n - i

$(n-1) + (n-2) + \ldots + (1)$

$= n(n-1)/2$

$\in O(n^2)$

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with $n >= 2$ )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j < a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$  is in increasing order}
```

Strategy: work from the inside out

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j$ < $a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$  is in increasing order}
```

O(1)

$t_1 + t_2$

Strategy: work from the inside out

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j < a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$  is in increasing order}
```

O(1)

Simple for loop, repeats n-i times } $O(n-i)$

worst case is when $i=1$

$O(n-1) = O(n).$

Strategy: work from the inside out

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j$ < $a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$  is in increasing order}
```

~~O(n-i),~~ O(n)
but i ranges from 1 to n-1

Strategy: work from the inside out

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
    m := i
    for j:= i+1 to n
        if ( $a_j$ < $a_m$ ) then m := j
    interchange $a_i$ and $a_m$

{ $a_1$, ..., $a_n$  is in increasing order}
```

Worst case: when i =1, O(n)

Strategy: work from the inside out

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
     procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
     for i := 1 to n-1
O(1)   m := i
         O(n)
O(1)   interchange $a_i$ and $a_m$
     { $a_1$, ..., $a_n$  is in increasing order}
```

$(n-1)O(n) = O(n^2)$

Strategy: work from the inside out

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
        O(n)

{ $a_1$, ..., $a_n$  is in increasing order}
```

Strategy: work from the inside out

Selection Sort (MinSort) Pseudocode

Now, straight to big O

```
procedure selection sort($a_1$, $a_2$, ..., $a_n$: real numbers with n >=2 )
for i := 1 to n-1
        O(n)

{ $a_1$, ..., $a_n$  is in increasing order}
```

Nested for loop, repeats O(n) times

Total: O(n^2)

Strategy: work from the inside out