16.1 Searching and algorithms

Algorithms and linear search

An ***algorithm*** is a sequence of steps for accomplishing a task. ***Linear search*** is a search algorithm that starts from the beginning of a list, and checks each element until the search key is found or the end of the list is reached.

PARTICIPATION ACTIVITY 16.1.1: Linear search algorithm checks each element until key is found.

Animation captions:

1. Linear search starts at first element and searches elements one-by-one.
2. Linear search will compare all elements if the search key is not present.

Figure 16.1.1: Linear search algorithm.

```
def linear_search(numbers, key):
    for i in range(len(numbers)):
        if numbers[i] == key:
            return i
    return -1   # not found


numbers = [2, 4, 7, 10, 11, 32, 45, 87]
print('NUMBERS:', end=' ')
for num in numbers:
    print(str(num), end=' ')
print()

key = int(input('Enter a value: '))
key_index = linear_search(numbers, key)

if key_index == -1:
    print(str(key) + ' was not found.')
else:
    print('Found ' + str(key) + ' at index ' + str(key_index) + '.')
```



```
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 10
Found 10 at index 3.
...
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 17
17 was not found.
```

PARTICIPATION ACTIVITY 16.1.2: Linear search algorithm execution.

Given list: [20, 4, 114, 23, 34, 25, 45, 66, 77, 89, 11].

1) How many list elements will be compared to find 77 using linear search?

Check Show answer

2) How many list elements will be checked to find the value 114 using linear search?

Check Show answer

3) How many list elements will be checked if the search key is not found using linear search?

Check Show answer

Algorithm runtime

An algorithm's ***runtime*** is the time the algorithm takes to execute. If each comparison takes 1 µs (1 microsecond), a linear search algorithm's runtime is up to 1 s to search a list with 1,000,000 elements, 10 s for 10,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, could require more than 3 minutes.

An algorithm typically uses a number of steps proportional to the size of the input. For a list with 32 elements, linear search requires at most 32 comparisons: 1 comparison if the search key is found at index 0, 2 if found at index 1, and so on, up to 32 comparisons if the the search key is not found. For a list with N elements, linear search thus requires at most N comparisons. The algorithm is said to require "on the order" of N comparisons.

PARTICIPATION ACTIVITY 16.1.3: Linear search runtime.

1) Given a list of 10,000 elements, and if each comparison takes 2 µs, what is the fastest possible runtime for linear search?

µs

Check Show answer

2) Given a list of 10,000 elements, and if each comparison takes 2 µs, what is the longest possible runtime for linear search?

µs

16.2 Binary search

Using binary search

Linear search may require searching all list elements, which can lead to long runtimes. For example, searching for a contact on a smartphone one-by-one from first to last can be time consuming. Because a contact list is sorted, a faster search, known as binary search, checks the middle contact first. If the desired contact comes alphabetically before the middle contact, binary search will then search the first half and otherwise the last half. Each step reduces the contacts that need to be searched by half.

PARTICIPATION ACTIVITY 16.2.1: Using binary search to search contacts on your phone.

Animation captions:

1. A contact list stores contacts sorted by name. Searching for Pooja using a binary search starts by checking the middle contact.
2. The middle contact is Muhammad. Pooja is alphabetically after Muhammad, so the binary search only searches the contacts after Muhammad. Only half the contacts now need to be searched.
3. Binary search continues by checking the middle element between Muhammad and the last contact. Pooja is before Sharod, so the search continues with only those contacts between Muhammad and Sharod, which is one fourth of the contacts.
4. The middle element between Muhammad and Sharod is Pooja. Each step reduces the number of contacts to search by half.

PARTICIPATION ACTIVITY 16.2.2: Using binary search to search a contact list.

A contact list is searched for Bob.
Assume the following contact list: Amy, Bob, Chris, Holly, Ray, Sarah, Zoe

1) What is the first contact searched?

Check Show answer

2) What is the second contact searched?

Check Show answer

Binary search algorithm

Binary search is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as a list). Binary search first checks the middle element of the list. If the search key is found, the algorithm returns the matching location. If the search key is not found, the algorithm repeats the search on the remaining left sublist (if the search key was less than the middle element) or the remaining right sublist (if the search key was greater than the middle element).

PARTICIPATION ACTIVITY 16.2.3: Binary search efficiently searches sorted list by reducing the search space by half each iteration.

Animation captions:

1. Elements with indices between low and high remain to be searched.
2. Search starts by checking the middle element.
3. If search key is greater than element, then only elements in right sublist need to be searched.
4. Each iteration reduces search space by half. Search continues until key found or search space is empty.

Figure 16.2.1: Binary search algorithm.

```python
def binary_search(numbers, key):
    low = 0
    high = len(numbers) - 1

    while high >= low:
        mid = (high + low) // 2
        if numbers[mid] < key:
            low = mid + 1
        elif numbers[mid] > key:
            high = mid - 1
        else:
            return mid
    return -1 # not found


numbers = [2, 4, 7, 10, 11, 32, 45, 87]
print('NUMBERS:', end=' ')
for num in numbers:
    print(num, end=' ')
print()

key = int(input('Enter a value: '))
key_index = binary_search(numbers, key)

if key_index == -1:
    print(str(key) + ' was not found.')
else:
    print('Found ' + str(key) + ' at index ' + str(key_index) + '.')
```



```
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 10
Found 10 at index 3.
...
NUMBERS: 2 4 7 10 11 32 45 87
Enter a value: 17
17 was not found.
```

PARTICIPATION ACTIVITY 16.2.4: Binary search algorithm execution.

Given list: [4, 11, 17, 18, 25, 45, 63, 77, 89, 114].

1) How many list elements will be checked to find the value 77 using binary search?

Check **Show answer**

2)

How many list elements will be checked to find the value 17 using binary search?

Check Show answer

3) Given a list with 32 elements, how many list elements will be checked if the key is less than all elements in the list, using binary search?

Check Show answer

Binary search efficiency

Binary search is incredibly efficient in finding an element within a sorted list. During each iteration or step of the algorithm, binary search reduces the search space (i.e., the remaining elements to search within) by half. The search terminates when the element is found or the search space is empty (element not found). For a 32 element list, if the search key is not found, the search space is halved to have 16 elements, then 8, 4, 2, 1, and finally none, requiring only 6 steps. For an N element list, the maximum number of steps required to reduce the search space to an empty sublist is $\lfloor log_2 N \rfloor + 1$. Ex: $\lfloor log_2 32 \rfloor + 1 = 6$.

PARTICIPATION ACTIVITY 16.2.5: Speed of linear search versus binary search to find a number within a sorted list.

Animation captions:

©zyBooks 03/05/20 10:46 591419
Alexey Munishkin
UCSCCSE20NawabWinter2020

1. A binary search begins with the middle element of the list. Each subsequent search reduces the search space by half. Using binary search, a match was found with only 3 comparisons.
2. Using linear search, a match was found after 6 comparisons. Compared to a linear search, binary search is incredibly efficient in finding an element within a sorted list.

If each comparison takes 1 μs (1 microsecond), a binary search algorithm's runtime is at most 20 μs to search a list with 1,000,000 elements, 21 μs to search 2,000,000 elements, 22 μs to search 4,000,000 elements, and so on. Ex: Searching Amazon's online store, which has more than 200 million items, requires less than 28 μs; up to 7,000,000 times faster than linear search.

PARTICIPATION ACTIVITY 16.2.6: Linear and binary search runtime.

Answer the following questions assuming each comparison takes 1 μs.

1) Given a list of 1024 elements, what is the runtime for linear search if the search key is less than all elements in the list?

μs

Check Show answer

2) Given a list of 1024 elements, what is the runtime for binary search if the search key is less than all elements in the list?

μs

Check Show answer

16.3 O notation

Big O notation

Big O notation is a mathematical way of describing how a function (running time of an algorithm) generally behaves in relation to the input size. In Big O notation, all functions that have the same growth rate (as determined by the highest order term of the function) are characterized using the same Big O notation. In essence, all functions that have the same growth rate are considered equivalent in Big O notation.

Given a function that describes the running time of an algorithm, the Big O notation for that function can be determined using the following rules:

1. If f(x) is a sum of several terms, the highest order term (the one with the fastest growth rate) is kept and others are discarded.
2. If f(x) has a term that is a product of several factors, all constants (those that are not in terms of x) are omitted.

PARTICIPATION ACTIVITY 16.3.1: Determining Big O notation of a function.

Animation captions:

1. Determine a function that describes the running time of the algorithm, and then compute the Big O notation of that function.
2. Apply rules to obtain the Big O notation of the function.
3. All functions with the same growth rate are considered equivalent in Big O notation.

PARTICIPATION ACTIVITY 16.3.2: Big O notation.

1) Which of the following Big O notations is equivalent to O(N+9999)?

- O(1)
- O(N)
- O(9999)

2) Which of the following Big O notations is equivalent to O(734·N)?

- O(N)
- O(734)
- O(734·N^2)

3) Which of the following Big O notations is equivalent to $O(12 \cdot N + 6 \cdot N^3 + 1000)$?

- $O(1000)$
- $O(N)$
- $O(N^3)$

The following rules are used to determine the Big O notation of composite functions: c denotes a constant

Figure 16.3.1: Rules for determining Big O notation of composite functions.

Composite function	Big O notation
$c \cdot O(f(x))$	$O(f(x))$
$c + O(f(x))$	$O(f(x))$
$g(x) \cdot O(f(x))$	$O(g(x) \cdot O(f(x)))$
$g(x) + O(f(x))$	$O(g(x) + O(f(x)))$

PARTICIPATION ACTIVITY 16.3.3: Big O notation for composite functions.

Determine the simplified Big O notation.

1) $10 \cdot O(N^2)$

- $O(10)$
- $O(N^2)$
- $O(10 \cdot N^2)$

2) $10 + O(N^2)$

- $O(10)$

O(N^2)

- O($10 + N^2$)

3) $3 \cdot N \cdot O(N^2)$

- O(N^2)
- O($3 \cdot N^2$)
- O(N^3)

4) $2 \cdot N^3 + O(N^2)$

- O(N^2)
- O(N^3)
- O($N^2 + N^3$)

5) log_2N

- O(log_2N)
- O($logN$)
- O(N)

Runtime growth rate

One consideration in evaluating algorithms is that the efficiency of the algorithm is most critical for large input sizes. Small inputs are likely to result in fast running times because N is small, so efficiency is less of a concern. The table below shows the runtime to perform f(N) instructions for different functions f and different values of N. For large N, the difference in computation time varies greatly with the rate of growth of the function f. The data assumes that a single instruction takes 1 μs to execute.

Table 16.3.1: Growth rates for different input sizes.

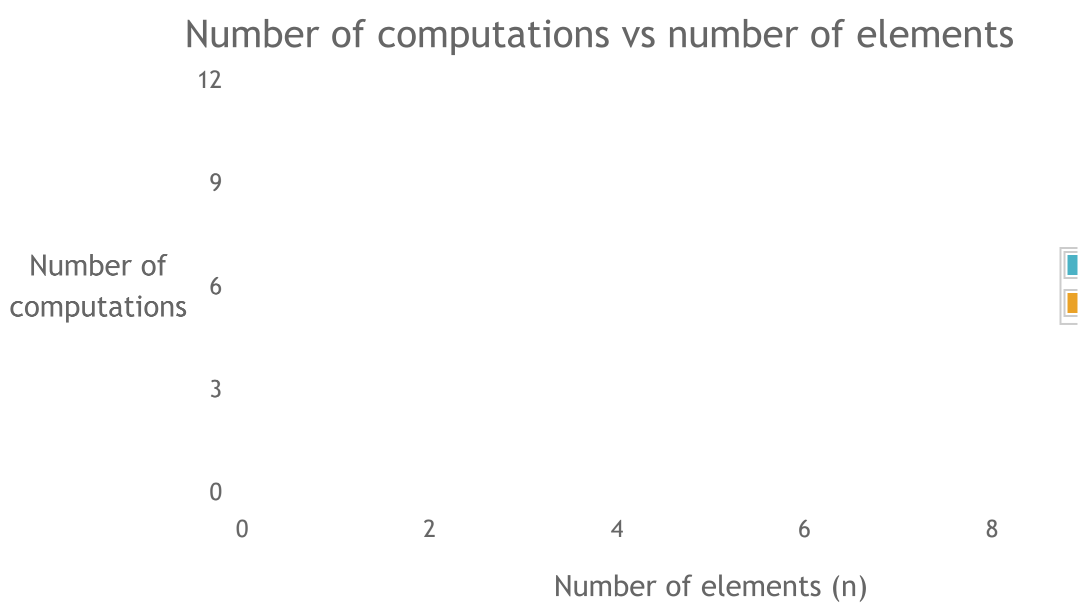
Function	N = 10	N = 50	N = 100	N = 1000	N = 10000	N = 100000
log_2N	3.3 μs	5.65 μs	6.6 μs	9.9 μs	13.3 μs	16.6 μs
N	10 μs	50 μs	100 μs	1000 μs	10 ms	1 s
$Nlog_2N$	.03 ms	.28 ms	.66 ms	.099 s	.132 s	1.66 s

N^2	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
N^3	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years
2^N	.001 s	35.7 years	*	*	*	*

The interactive tool below illustrates graphically the growth rate of commonly encountered functions.

PARTICIPATION ACTIVITY

16.3.4: Computational complexity graphing tool.



Common Big O complexities

Many commonly used algorithms have running time functions that belong to one of a handful of growth functions. These common Big O notations are summarized in the following table. The table shows the Big O notation, the common word used to describe algorithms that belong to that notation, and an example with source code. Clearly, the best algorithm is one that has constant time complexity. Unfortunately, not all problems can be solved using constant complexity algorithms. In fact, in many cases, computer scientists have proven that certain types of problems can only be solved using quadratic or exponential algorithms.

Figure 16.3.2: Runtime complexities for various code examples.

<table>
<tr><th>Notation</th><th>Name</th><th>Example Python</th></tr>
<tr><td>O(1)</td><td>Constant</td><td><pre>
def find_min(x, y):
   if x < y:
      return x
   else:
      return y
</pre></td></tr>
<tr><td>O(log N)</td><td>Logarithmic</td><td><pre>
def binary_search(numbers, key):
   low = 0
   high = len(numbers) - 1

   while high >= low:
      mid = (high + low) // 2
      if numbers[mid] < key:
         low = mid + 1
      elif numbers[mid] > key:
         high = mid - 1
      else:
         return mid
   return -1 # not found
</pre></td></tr>
<tr><td>O(N)</td><td>Linear</td><td><pre>
def linear_search(numbers, key):
    for i in range(len(numbers)):
        if numbers[i] == key:
             return i
    return -1 # not found
</pre></td></tr>
<tr><td>O(N log N)</td><td>Log-linear</td><td><pre>
def merge_sort(numbers, i, k):
   if i < k:
      j = (i + k) // 2   # Find midpoint
      merge_sort(numbers, i, j)   # Sort left part
      merge_sort(numbers, j + 1, k) # Sort right part
      merge(numbers, i, j, k)   # Merge parts
</pre></td></tr>
<tr><td>O(N²)</td><td>Quadratic</td><td></td></tr>
</table>

		``def selection_sort(numbers):
		    for i in range(len(numbers)):
		        index_smallest = i
		        for j in range(i + 1, len(numbers)):
		            if numbers[j] < numbers[index_smallest]:
		                index_smallest = j
		
		        temp = numbers[i]
		        numbers[i] = numbers[index_smallest]
		        numbers[index_smallest] = temp```

$O(c^N)$	Exponential	```def fibonacci(N):
		    if (1 == N) or (2 == N):
		        return
		    return fibonacci(N-1) + fibonacci(N-2)```

PARTICIPATION ACTIVITY 16.3.5: Big O notation and growth rates.

1) O(5) has a _____ runtime complexity.

- constant
- linear
- exponential

2) O(N log N) has a _____ runtime complexity.

- constant
- log-linear
- logarithmic

3) $O(N + N^2)$ has a _____ runtime complexity.

- linear-quadratic
- exponential
- quadratic

4) A linear search has a _____ runtime complexity.

- O(log N)

- O(N)
- O(N^2)

5) A selection sort has a _____ runtime complexity.

- O(N)
- O(N log N)
- O(N^2)

16.4 Algorithm analysis

Worst-case algorithm analysis

To analyze how runtime of an algorithm scales as the input size increases, we first determine how many operations the algorithm executes for a specific input size, N. Then, the big-O notation for that function is determined. Algorithm runtime analysis often focuses on the worst-case runtime complexity. The ***worst-case runtime*** of an algorithm is the runtime complexity for an input that results in the longest execution. Other runtime analyses include best-case runtime and average-case runtime. Determining the average-case runtime requires knowledge of the statistical properties of the expected data inputs.

PARTICIPATION ACTIVITY 16.4.1: Runtime analysis: Finding the max value.

Animation content:

`undefined`

Animation captions:

1. Runtime analysis determines the total number of operations. Operations include assignment, addition, comparison, etc.
2. The for loop iterates N times, in this case size of list numbers.

3. For each loop iteration, num is assigned the next value in the list. In the worst-case, the if's expression is true, resulting in 2 operations.
4. The function f(N) specifies the number of operations executed for input size N. The big-O notation for the function is the algorithm's worst-case runtime complexity.

PARTICIPATION ACTIVITY 16.4.2: Worst-case runtime analysis.

1) Which function best represents the number of operations in the worst-case?

```
sum = 0
for num in numbers:
    sum = sum + num
```

- f(N) = 2N
- f(N) = 2N + 1
- f(N) = 2 + N (N + 1)

2) What is the big-O notation for the worst-case runtime?

```
neg_count = 0
for num in numbers:
    if num < 0:
        neg_count = neg_count +
1
```

- f(N) = 1 + 3N
- O(3N + 1)
- O(N)

3) What is the big-O notation for the worst-case runtime?

```
for i in range(N):
    if i % 2 == 0:
        out_val[i] = in_vals[i]
* i
```


- O(1)
- O($\frac{N}{2}$)
- O(N)

4) What is the big-O notation for the worst-case runtime?

```
n_val = N
steps = 0
while n_val > 0:
    n_val = n_val / 2
    steps = steps + 1
```

- O(log N)
- O($\frac{N}{2}$)
- O(N)

5) What is the big-O notation for the **best-case** runtime?

```
below_min_sum = 0.0
below_min_count = 0
while (i < len(numbers)) and
(numbers[i] <= max_val):
    below_min_count =
below_min_count + 1
    below_min_sum = numbers[i]
    i = i + 1
avg_below = below_min_sum /
below_min_count
```

- O(1)
- O(N)

Counting constant time operations

For algorithm analysis, the definition of a single operation does not need to be precise. An operation can be any statement (or constant number of statements) that has a constant runtime complexity, O(1). Since constants are omitted in big-O notation, any constant number of constant time operations is O(1). So, precisely counting the number of constant time operations in a finite sequence is not needed. Ex: An algorithm with a single loop that executes 5 operations before the loop, 3 operations in each loop iteration, and 6 operations after the loop would have a runtime of f(N) = 5 + 3N + 6, which can be written as O(1) + O(N) + O(1) = O(N). If the number of operations before the loop was 100, the big-O notation for those operation is still O(1).

PARTICIPATION ACTIVITY 16.4.3: Simplified runtime analysis: A constant number of constant time operations is O(1).

Animation content:

undefined

Animation captions:

1. Constants are omitted in big-O notation, so any constant number of constant time operations is O(1).
2. The for loop iterates N times. Big-O complexity can be written as a composite function and simplified.

PARTICIPATION ACTIVITY 16.4.4: Constant time operations.

1) A for loop of the form `for num in numbers:` that does not have nested loops or function calls, and does not modify num in the loop will always has a complexity of O(N).

- True
- False

2) The complexity of the algorithm below is O(1).

```
if time_hour < 6:
    toll_amount = 1.55
elif time_hour < 10:
    toll_amount = 4.65
elif time_hour < 18:
    toll_amount = 2.35
else:
    toll_amount = 1.55
```

- True
- False

3)

The complexity of the algorithm below is O(1).

```
for i in range(24):
    if time_hour < 6:
        toll_schedule[i] = 1.55
    elif time_hour < 10:
        toll_schedule[i] = 4.65
    elif time_hour < 18:
        toll_schedule[i] = 2.35
    else:
        toll_schedule[i] = 1.55
```


- True
- False

Runtime analysis of nested loops

Runtime analysis for nested loops requires summing the runtime of the inner loop over each outer loop iteration. The resulting summation can be simplified to determine the big-O notation.

PARTICIPATION ACTIVITY 16.4.5: Runtime analysis of nested loop: Selection sort algorithm.

Animation content:

```
undefined
```

Animation captions:

1. For each iteration of the outer loop, the runtime of the inner loop is determined and added together to form a summation. For iteration i = 0, the inner loop executes N - 1 iterations.
2. For i = 1, the inner loop iterates N - 2 times: iterating from j = 2 to N - 1.
3. For i = N - 2, the inner loop iterates once: iterating from j = N - 1 to N - 1.
4. The summation is the sum of a consecutive sequence of numbers, and can be simplified.
5. Each iteration of the loops requires a constant number of operations, which is defined as the constant c.
6. Additionally, each iteration of the outer loop requires a constant number of operations, which is defined as the constant d.
7. Big-O notation omits the constant values, and the runtime is equal to the summation of the total inner loop iterations.

Figure 16.4.1: Common summation: Summation of consecutive numbers.

$$(N-1)+(N-2)+\cdots+2+1=\frac{N(N-1)}{2}=O(N^2)$$

PARTICIPATION ACTIVITY 16.4.6: Nested loops.

Determine the big-O worst-case runtime for each algorithm. For each algorithm, `N = len(numbers)`.

1)
```
for i in range(N):
    for j in range(N):
        if numbers[i] < numbers[j]:
            eq_perms = eq_perms + 1
        else:
            neq_perms = neq_perms + 1
```

- O(N)
- O(N^2)

2)
```
for i in range(N):
    for j in range(N - 1):
        if numbers[j + 1] < numbers[j]:
            temp = numbers[j]
            numbers[j] = numbers[j + 1]
            numbers[j + 1] = temp
```

- O(N)
- O(N^2)

3)
```
for i in range(0, N, 2):
    for j in range(0, N, 2):
        c_vals[i][j] = in_vals[i] * j
```

- O(N)
- O(N^2)

4)
```
for i in range(N):
    for j in range(i, N - 1):
        c_vals[i][j] =
in_vals[i] * j
```

- O(N^2)
- O(N^3)

5)
```
for i in range(N):
    sum = 0
    for j in range(N):
        for k in range(N):
            sum = sum +
a_vals[i][k] * b_vals[k][j]
        c_vals[i][j] = sum
```

- O(N)
- O(N^2)
- O(N^3)

16.5 Sorting: Introduction

Sorting is the process of converting a list of elements into ascending (or descending) order. For example, given a list of numbers [17, 3, 44, 6, 9], the list after sorting is [3, 6, 9, 17, 44]. You may have carried out sorting when arranging papers in alphabetical order, or arranging envelopes to have ascending zip codes (as required for bulk mailings).

The challenge of sorting is that a program can't "see" the entire list to know where to move an element. Instead, a program is limited to simpler steps, typically observing or swapping just two elements at a time. So sorting just by swapping values is an important part of sorting algorithms.

Note that a Python programmer could of course make use of the `sort()` list method, or `sorted()` builtin function. This section describes what the implementation of those functions might look like.

PARTICIPATION ACTIVITY 16.5.1: Sort by swapping tool.

Sort the numbers from smallest on left to largest on right. Select two numbers then click "Swap values".

PARTICIPATION ACTIVITY 16.5.2: Sorted elements.

1) The list is sorted into ascending order:
[3, 9, 44, 18, 76]

- True
- False

2) The list is sorted into descending order:
[20, 15, 10, 5, 0]

- True
- False

3) The list is sorted into descending order:
[99.87, 99.02, 67.93, 44.10]

- True
- False

4) The list is sorted into descending order:
[F, D, C, B, A]

- True
- False

5) The list is sorted into ascending order:
['chopsticks', 'forks', 'knives', 'spork']

- True
- False

6)

The list is sorted into ascending order:
['great', 'greater', 'greatest']

- True
- False

16.6 Selection sort

Selection sort algorithm

Selection sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly selects the proper next value to move from the unsorted part to the end of the sorted part.

PARTICIPATION ACTIVITY 16.6.1: Selection sort.

Animation content:

`undefined`

Animation captions:

1. Selection sort treats the input as two parts, a sorted and unsorted part. Variables i and j keep track of the two parts.
2. The selection sort algorithm searches the unsorted part of the array for the smallest element; index_smallest stores the index of the smallest element found.
3. Elements at i and index_smallest are swapped.
4. Indices for the sorted and unsorted parts are updated.
5. The unsorted part is searched again, swapping the smallest element with the element at i.
6. The process repeats until all elements are sorted.

The index variable i denotes the dividing point. Elements to the left of i are sorted, and elements including and to the right of i are unsorted. All elements in the unsorted part are searched to find the index of the element with the smallest value. The variable index_smallest stores the index of the smallest element in the unsorted part. Once the element with the smallest value is found, that element is swapped with the element at location i. Then, the index i is advanced one place to the right, and the process repeats.

The term "selection" comes from the fact that for each iteration of the outer loop, a value is selected for position i.

PARTICIPATION ACTIVITY 16.6.2: Selection sort algorithm execution.

Assume selection sort's goal is to sort in ascending order.

1) Given list [9, 8, 7, 6, 5], what value will be in the 0^{th} element after the first pass over the outer loop (i = 0)?

Check Show answer

2) Given list [9, 8, 7, 6, 5], how many swaps will occur during the first pass of the outer loop (i = 0)?

Check Show answer

3) Given list [5, 9, 8, 7, 6] and i = 1, what will be the list after completing the second outer loop iteration? Use brackets in your answer, e.g., "[1, 2, 3]".

Check Show answer

Advantages of selection sort

Selection sort has the advantage of being easy to code, involving one loop nested within another loop, as shown below.

Figure 16.6.1: Selection sort algorithm.


```
def selection_sort(numbers):
    for i in range(len(numbers) - 1):
        # Find index of smallest remaining element
        index_smallest = i
        for j in range(i + 1, len(numbers)):
            if numbers[j] < numbers[index_smallest]:
                index_smallest = j

        # Swap numbers[i] and numbers[index_smallest]
        temp = numbers[i]
        numbers[i] = numbers[index_smallest]
        numbers[index_smallest] = temp


numbers = [10, 2, 78, 4, 45, 32, 7, 11]
print('UNSORTED:', end=' ')
for num in numbers:
    print(str(num), end=' ')
print()

selection_sort(numbers)
print('SORTED:', end=' ')
for num in numbers:
    print(str(num), end=' ')
print()
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

Selection sort may require a large number of comparisons. The selection sort algorithm runtime is O(N^2). If a list has N elements, the outer loop executes N - 1 times. For each of those N - 1 outer loop executions, the inner loop executes an average of $\frac{N}{2}$ times. So the total number of comparisons is proportional to $(N-1) \cdot \frac{N}{2}$, or O(N^2). Other sorting algorithms involve more complex algorithms but have faster execution times.

PARTICIPATION ACTIVITY 16.6.3: Selection sort runtime.

1) When sorting a list with 50 elements, `index_smallest` will

be assigned to a minimum of
_____ times.

Check Show answer

2) How many times longer will sorting a list of 20 elements take compared to sorting a list of 10 elements?

Check Show answer

3) How many times longer will sorting a list of 500 elements take compared to a list of 50 elements?

Check Show answer

16.7 Insertion sort

Insertion sort algorithm

Insertion sort is a sorting algorithm that treats the input as two parts, a sorted part and an unsorted part, and repeatedly inserts the next value from the unsorted part into the correct location in the sorted part.

PARTICIPATION ACTIVITY 16.7.1: Insertion sort.

Animation captions:

1. Insertion sort treats the input as two parts, a sorted and unsorted part. Variable i is the index of the first unsorted element. Initially, the element at index 0 is assumed to be sorted, so i starts at 1.
2. Variable j keeps track of the index of the current element being inserted into the sorted part. If the current element is less than the element to the left, the values are swapped.
3. Once the current element is inserted in the correct location in the sorted part, i is incremented to the next element in the unsorted part.
4. If the current element being inserted is smaller than all elements in the sorted part, that element will be repeatedly swapped with each sorted element until index 0 is reached.
5. Once all elements in the unsorted part are inserted in the sorted part, the list is sorted.

Figure 16.7.1: Insertion sort algorithm.

```
def insertion_sort(numbers):
    for i in range(1, len(numbers)):
        j = i
        # Insert numbers[i] into sorted part
        # stopping once numbers[i] in correct position
        while j > 0 and numbers[j] < numbers[j - 1]:
            # Swap numbers[j] and numbers[j - 1]
            temp = numbers[j]
            numbers[j] = numbers[j - 1]
            numbers[j - 1] = temp
            j = j - 1


numbers = [10, 2, 78, 4, 45, 32, 7, 11]
print ('UNSORTED:', end=' ')
for num in numbers:
    print (str(num), end=' ')
print()

insertion_sort(numbers)
print ('SORTED:', end=' ')
for num in numbers:
    print (str(num), end=' ')
print()
```



```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

Insertion sort execution

The index variable i denotes the starting position of the current element in the unsorted part. Initially, the first element (i.e., element at index 0) is assumed to be sorted, so the outer for loop initializes i to 1. The inner while loop inserts the current element into the sorted part by repeatedly swapping the current element with the elements in the sorted part that are larger. Once a smaller or equal element is found in sorted part, the current element has been inserted in the correct location and the while loop terminates.

PARTICIPATION ACTIVITY 16.7.2: Insertion sort algorithm execution.

Assume insertion sort's goal is to sort in ascending order.

1) Given list [20, 14, 85, 3, 9], what value will be in the 0^{th} element after the first pass over the outer loop (i = 1)?

Check Show answer

2) Given list [10, 20, 6, 14, 7], what will be the list after completing the second outer loop iteration (i = 2)? Use brackets in your answer, e.g. "[1, 2, 3]".

Check Show answer

3) Given list [1, 9, 17, 18, 2], how many swaps will occur during the outer loop execution (i = 4)?

Check Show answer

Insertion sort runtime

Insertion sort's typical runtime is $O(N^2)$. If a list has N elements, the outer loop executes N - 1 times. For each outer loop execution, the inner loop may need to examine all elements in the sorted part. Thus, the inner loop executes on average $\frac{N}{2}$ times. So the total number of comparisons is proportional to $(N-1) \cdot (\frac{N}{2})$, or $O(N^2)$. Other sorting algorithms involve more complex algorithms but faster execution.

PARTICIPATION ACTIVITY 16.7.3: Insertion sort runtime.

1) In the worst case, assuming each comparison takes 1 µs, how long will insertion sort algorithm take to sort a list of 10 elements?

µs

Check Show answer

2) Using the Big O runtime complexity, how many times longer will sorting a list of 20 elements take compared to sorting a list of 10 elements?

Check Show answer

Insertion sort for nearly sorted lists

For sorted or nearly sorted inputs, insertion sort's runtime is O(N). A ***nearly sorted*** list only contains a few elements not in sorted order. Ex: [4, 5, 17, 25, 89, 14] is nearly sorted having only one element not in sorted position.

PARTICIPATION ACTIVITY 16.7.4: Nearly sorted lists.

Determine if each of the following lists is unsorted, sorted, or nearly sorted. Assume ascending order.

1) [6, 14, 85, 102, 102, 151]

- Unsorted
- Sorted
- Nearly sorted

2) [23, 24, 36, 48, 19, 50, 101]

- Unsorted
- Sorted
- Nearly sorted

3) [15, 19, 21, 24, 2, 3, 6, 11]

- Unsorted
- Sorted
- Nearly sorted

For each outer loop execution, if the element is already in sorted position, only a single comparison is made. Each element not in sorted position requires at most N comparisons. If there are a constant number, C, of unsorted elements, sorting the N - C sorted elements requires one comparison each, and sorting the C unsorted elements requires at most N comparisons each. The runtime for nearly sorted inputs is O((N - C) * 1 + C * N) = O(N).

PARTICIPATION ACTIVITY 16.7.5: Using insertion sort for nearly sorted list.

Animation captions:

1. Unsorted part initially contains the first element.
2. An element already in sorted position only requires a single comparison, which is O(1) complexity.
3. An element not in sorted position requires O(N) comparisons. For nearly sorted inputs, insertion sort's runtime is O(N).

PARTICIPATION ACTIVITY

16.7.6: Insertion sort algorithm execution for nearly sorted input.

Assume insertion sort's goal is to sort in ascending order.

1) Given list [10, 11, 12, 13, 14, 5], how many comparisons will be made during the third outer loop execution (i = 3)?

Check Show answer

2) Given list [10, 11, 12, 13, 14, 7], how many comparisons will be made during the final outer loop execution (i = 5)?

Check Show answer

3) Given list [18, 23, 34, 75, 3], how many total comparisons will insertion sort require?

Check Show answer

16.8 Quicksort

Quicksort and partitioning

Quicksort is a sorting algorithm that repeatedly partitions the input into low and high parts (each part unsorted), and then recursively sorts each of those parts. To partition the input, quicksort chooses a pivot to divide the data into low and high parts. The ***pivot*** can be any value within the array being sorted, commonly the value of the middle array element. Ex: For the list [4, 34, 10, 25, 1], the middle element is located at index 2 (the middle of indices 0..4) and has a value of 10.

To partition the input, the quicksort algorithm divides the array into two parts, referred to as the low partition and the high partition. All values in the low partition are less than or equal to the pivot value. All values in the high partition are greater than or equal to the pivot value. The values in each partition are not necessarily sorted. Ex: Partitioning [4, 34, 10, 25, 1] with a pivot value of 10 results in a low partition of [4, 10, 1] and a high partition of [34, 25]. Values equal to the pivot may appear in either or both of the partitions.

PARTICIPATION ACTIVITY

16.8.1: Quicksort partitions data into a low part with data less than/equal to a pivot value and a high part with data greater than/equal to a pivot value.

Animation content:

```
undefined
```

Animation captions:

1. The pivot value is the value of the middle element.
2. Index l begins at element i and is incremented until a value greater than the pivot is found.
3. Index h begins at element k, and is decremented until a value less than the pivot is found.
4. Elements at indices l and h are swapped, moving those elements to the correct partitions.
5. The partition process repeats until indices l and h reach or pass each other (l >= h), indicating all elements have been partitioned.
6. Once partitioned, the algorithm returns h indicating the highest index of the low partition. The partitions are not yet sorted.

The partitioning algorithm uses two index variables l and h (low and high), initialized to the left and right sides of the current elements being sorted. As long as the value at index l is less than the pivot value, the algorithm increments l, because the element should remain in the low partition. Likewise, as long as the value at index h is greater than the pivot value, the algorithm decrements h, because the element should remain in the high partition. Then, if l >= h, all elements have been partitioned, and the partitioning algorithm returns h, which is the index of the last element in the low partition. Otherwise, the elements at indices l and h are swapped to move those elements to the correct partitions. The algorithm then increments l, decrements h, and repeats.

PARTICIPATION ACTIVITY 16.8.2: Quicksort pivot location and value.

Determine the midpoint and pivot values.

1) numbers = [1, 2, 3, 4, 5], i = 0, k = 4

`midpoint =`

Check **Show answer**

2) numbers = [1, 2, 3, 4, 5], i = 0, k = 4

`pivot =`

Check **Show answer**

3) numbers = [200, 11, 38, 9], i = 0, k = 3

`midpoint =`

Check **Show answer**

4) numbers = [200, 11, 38, 9], i = 0, k = 3

`pivot =`

Check **Show answer**

5) numbers = [55, 7, 81, 26, 0, 34, 68, 125], i = 3, k = 7

midpoint =

Check **Show answer**

6) numbers = [55, 7, 81, 26, 0, 34, 68, 125], i = 3, k = 7

pivot =

Check **Show answer**

PARTICIPATION ACTIVITY 16.8.3: Low and high partitions.

Determine if the low and high partitions are correct given h and pivot.

1) pivot = 35

		h		
1	4	35	62	98

- True
- False

2) pivot = 65

			h	
29	17	65	39	84

- True
- False

3) pivot = 5

	h					
5	13	16	77	84	20	19

- True
- False

4) pivot = 8

h

8	3	8	41	57	8

- True
- False

Quicksort algorithm and runtime

Once partitioned, each partition needs to be sorted. Quicksort is typically implemented as a recursive algorithm using calls to quicksort to sort the low and high partitions. This recursive sorting process continues until a partition has one or zero elements, and thus already sorted.

PARTICIPATION ACTIVITY 16.8.4: Quicksort.

Animation content:

`undefined`

Animation captions:

1. List contains more than one element. Partition the list.
2. Recursively call quicksort on the low and high partitions.
3. Low partition contains more than one element. Partition the low partition and recursively call quicksort.
4. Low partition contains one element, so partition is already sorted. High partition contains one element, so partition is already sorted.
5. High partition contains more than one element. Partition the high partition and recursively call quicksort.
6. Low partition contains more than one element. Partition the low partition and recursively call quicksort.
7. Low partition contains one element, so partition is already sorted. High partition contains one element, so partition is already sorted.
8. High partition contains one element, so partition is already sorted.
9. All elements are sorted.

Below is the recursive quicksort algorithm, including its key component the partitioning
.

Figure 16.8.1: Quicksort algorithm.

```python
def partition(numbers, i, k):
    #  Pick middle element as pivot
    midpoint = i + (k - i) // 2
    pivot = numbers[midpoint]

    #  Initialize variables
    done = False
    l = i
    h = k
    while not done:
        #  Increment l while numbers[l] < pivot
        while numbers[l] < pivot:
            l = l + 1
        #  Decrement h while pivot < numbers[h]
        while pivot < numbers[h]:
            h = h - 1
        """  If there are zero or one items remaining,
              all numbers are partitioned. Return h """
        if l >= h:
            done = True
        else:
            """  Swap numbers[l] and numbers[h],
                  update l and h """
            temp = numbers[l]
            numbers[l] = numbers[h]
            numbers[h] = temp
            l = l + 1
            h = h - 1
    return h


def quicksort(numbers, i, k):
    j = 0
    """  Base case: If there are 1 or zero entries to sort,
          partition is already sorted  """
    if i >= k:
        return
    """  Partition the data within the array. Value j returned
          from partitioning is location of last item in low partition. """
    j = partition(numbers, i, k)
    """  Recursively sort low partition (i to j) and
          high partition (j + 1 to k) """
    quicksort(numbers, i, j)
    quicksort(numbers, j + 1, k)
    return


numbers = [10, 2, 78, 4, 45, 32, 7, 11]
print ('UNSORTED:', end=' ')
for num in numbers:
    print (str(num), end=' ')
print()

#  Initial call to quicksort
quicksort(numbers, 0, len(numbers) - 1)
print ('SORTED:', end=' ')
for num in numbers:
    print (str(num), end=' ')
print()
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```

The following activity helps build intuition as to how partitioning a list into two unsorted parts, one part <= a pivot value and the other part >= a pivot value, and then recursively sorting each part, ultimately leads to a sorted list.

PARTICIPATION ACTIVITY 16.8.5: Quicksort tool.

Select all values in the current window that are less than the pivot for the left part, then press "Partition".

The quicksort algorithm's runtime is typically O(N log N). Quicksort has several partitioning levels , the first level dividing the input into 2 parts, the second into 4 parts, the third into 8 parts, etc. At each level, the algorithm does at most N comparisons moving the l and h indices. If the pivot yields two equal-sized parts, then there will be log N levels, requiring the N * log N comparisons.

PARTICIPATION ACTIVITY 16.8.6: Quicksort runtime.

Assume quicksort always chooses a pivot that divides the elements into two equal parts.

1) How many partitioning levels are required for a list of 8 elements?

Check **Show answer**

2) How many partitioning "levels" are required for a list of 1024 elements?

Check Show answer

3) How many total comparisons are required to sort a list of 1024 elements?

Check Show answer

For typical unsorted data, such equal partitioning occurs. However, partitioning may yield unequal sized part in some cases. If the pivot selected for partitioning is the smallest or largest element, one partition will have just 1 element, and the other partition will have all other elements. If this unequal partitioning happens at every level, there will be N - 1 levels, yielding $(N - 1) \cdot N)$, which is O(N^2). So the worst case runtime for the quicksort algorithm is O(N^2). Fortunately, this worst case runtime rarely occurs.

PARTICIPATION ACTIVITY 16.8.7: Worst case quicksort runtime.

Assume quicksort always chooses the smallest element as the pivot.

1) Given numbers = [7, 4, 2, 25, 19], i = 0, and k = 4, what is contents of the low partition? Use brackets in your answer, e.g., "[1, 2, 3]".

Check Show answer

2) How many partitioning "levels" of are required for a list of 5 elements?

Check Show answer

3) How many partitioning "levels" are required for a list of 1024 elements?

Check Show answer

4) How many total comparisons are required to sort a list of 1024 elements?

Check Show answer

16.9 Merge sort

Merge sort and partitioning

Merge sort is a sorting algorithm that divides a list into two halves, recursively sorts each half, and then merges the sorted halves to produce a sorted list. The recursive partitioning continues until a list of 1 element is reached, as a list of 1 element is already sorted.

PARTICIPATION ACTIVITY 16.9.1: Merge sort recursively divides the input into two halves, sorts each half, and merges the lists together.

Animation content:

```
undefined
```

Animation captions:

1. Merge sort recursively divides the list into two halves.
2. The list is divided until a list of 1 element is found.
3. A list of 1 element is already sorted.
4. At each level, the sorted lists are merged together while maintaining the sorted order.

The merge sort algorithm uses three index variables to keep track of the elements to sort for each recursive call. The index variable i is the index of the first element in the list, and the index variable k is the index of the last element. The index variable j is used to divide the list into two halves. Elements from i to j are in the left half, and elements from j + 1 to k are in the right half.

PARTICIPATION ACTIVITY 16.9.2: Merge sort partitioning.

Determine the index j and the left and right partitions.

1) numbers = [1, 2, 3, 4, 5], i = 0, k = 4

`j =`

Check Show answer

2) numbers = [1, 2, 3, 4, 5], i = 0, k = 4

`Left partition = [ ]`

Check Show answer

3) numbers = [1, 2, 3, 4, 5], i = 0, k = 4

`Right partition = [ ]`

Check Show answer

4) numbers = [34, 78, 14, 23, 8, 35],
i = 3, k = 5

```
j =
```

Check Show answer

5) numbers = [34, 78, 14, 23, 8, 35],
i = 3, k = 5

```
Left partition = [
]
```

Check Show answer

6) numbers = [34, 78, 14, 23, 8, 35],
i = 3, k = 5

```
Right partition = [
]
```

Check Show answer

Merge sort algorithm

Merge sort merges the two sorted partitions into a single list by repeatedly selecting the smallest element from either the left or right partition and adding that element to a temporary merged list. Once fully merged, the elements in the temporary merged list are copied back to the original list.

PARTICIPATION ACTIVITY

16.9.3: Merging partitions: Smallest element from the left or right partition is added one at a time to a temporary merged list. Once merged, the temporary list is copied back to the original list.

Animation captions:

1. Create a temporary list for merged numbers. Initialize mergePos, left_pos, and right_pos to the first element of each of the corresponding list.
2. Compare the element in the left and right partitions. Add the smallest value to the temporary list and update the relevant indices.

3. Continue to compare the elements in the left and right partitions until one of the partitions is empty.
4. If a partition is not empty, copy the remaining elements to the temporary list. The elements are already in sorted order.
5. Lastly, the elements in the temporary list are copied back to the original list.

PARTICIPATION ACTIVITY 16.9.4: Tracing merge operation.

Trace the merge operation by determining the next value added to merged_numbers.

14	18	35		17	38	49
0	1	2		3	4	5

1) left_pos = 0, right_pos = 3

Check Show answer

2) left_pos = 1, right_pos = 3

Check Show answer

3) left_pos = 1, right_pos = 4

Check Show answer

4) left_pos = 2, right_pos = 4

Check Show answer

5) left_pos = 3, right_pos = 4

Check Show answer

6) left_pos = 3, right_pos = 5

Check Show answer

Figure 16.9.1: Merge sort algorithm.

```
def merge(numbers, i, j, k):
    merged_size = k - i + 1    #  Size of merged partition
    merged_numbers = []         #  Temporary list for merged numbers
    for l in range(merged_size):
        merged_numbers.append(0)

    merge_pos = 0       #  Position to insert merged number

    left_pos = i        #  Initialize left partition position
    right_pos = j + 1   #  Initialize right partition position

    #  Add smallest element from left or right partition to merged numbers
    while left_pos <= j and right_pos <= k:
        if numbers[left_pos] < numbers[right_pos]:
            merged_numbers[merge_pos] = numbers[left_pos]
            left_pos = left_pos + 1
        else:
            merged_numbers[merge_pos] = numbers[right_pos]
            right_pos = right_pos + 1
        merge_pos = merge_pos + 1

    #  If left partition is not empty, add remaining elements to merged numbers
    while left_pos <= j:
        merged_numbers[merge_pos] = numbers[left_pos]
        left_pos = left_pos + 1
        merge_pos = merge_pos + 1

    #  If right partition is not empty, add remaining elements to merged
numbers
    while right_pos <= k:
        merged_numbers[merge_pos] = numbers[right_pos]
        right_pos = right_pos + 1
        merge_pos = merge_pos + 1

    #  Copy merge number back to numbers
    merge_pos = 0
    while merge_pos < merged_size:
        numbers[i + merge_pos] = merged_numbers[merge_pos]
        merge_pos = merge_pos + 1


def merge_sort(numbers, i, k):
    j = 0
    if i < k:
        j = (i + k) // 2  #  Find the midpoint in the partition

        #  Recursively sort left and right partitions
        merge_sort(numbers, i, j)
        merge_sort(numbers, j + 1, k)

        #  Merge left and right partition in sorted order
        merge(numbers, i, j, k)


numbers = [10, 2, 78, 4, 45, 32, 7, 11]
print ('UNSORTED:', end=' ')
for num in numbers:
    print (str(num), end=' ')
print()

#  initial call to merge_sort with index
```

```
merge_sort(numbers, 0, len(numbers) - 1)
print ('SORTED:', end=' ')
for num in numbers:
    print (str(num), end=' ')
print()
```

```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```


Merge sort runtime

The merge sort algorithm's runtime is O(N log N). Merge sort divides the input in half until a list of 1 element is reached, which requires log N partitioning levels. At each level, the algorithm does about N comparisons selecting and copying elements from the left and right partitions, yielding N * log N comparisons.

Merge sort requires O(N) additional memory elements for the temporary array of merged elements. For the final merge operation, the temporary list has the same number of elements as the input. Some sorting algorithms sort the list elements in place and require no additional memory, but are more complex to write and understand.

PARTICIPATION ACTIVITY 16.9.5: Merge sort runtime and memory complexity.

1) How many recursive partitioning levels are required for a list of 8 elements?

Check Show answer

2) How many recursive partitioning levels are required for a list of 2048 elements?

Check Show answer

3) How many elements will the temporary merge list have for

merging two partitions with 250 elements each?

Check Show answer