9.1 Searching and algorithms

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 9.1.1: 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 9.1.1: Linear search algorithm.

```
#include <iostream>
using namespace std;
int LinearSearch(int numbers[], int numbersSize, int key) {
   int i;

   for (i = 0; i < numbersSize; ++i) {
      if (numbers[i] == key) {
         return i;
      }
   }

   return -1; /* not found */
}
int main() {
   int numbers[] = { 2, 4, 7, 10, 11, 32, 45, 87 };
   const int NUMBERS_SIZE = 8;
   int i;
   int key;
   int keyIndex;

   cout << "NUMBERS: ";
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      cout << numbers[i] << ' ';
   }
   cout << endl;

   cout << "Enter a value: ";
   cin >> key;

   keyIndex = LinearSearch(numbers, NUMBERS_SIZE, key);

   if (keyIndex == -1) {
      cout << key << " was not found." << endl;
   }
   else {
      cout << "Found " << key << " at index " << keyIndex << "." << endl;
   }

   return 0;
}
```



```
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 9.1.2: 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

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 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 9.1.3: 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

Check Show answer

9.2 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 9.2.1: 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 9.2.2: 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)
- $O(734 \cdot N^2)$

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 9.2.1: 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 9.2.3: 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) $2 \cdot N^3 + O(N^2)$

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

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 9.2.1: Growth rates for different input sizes.

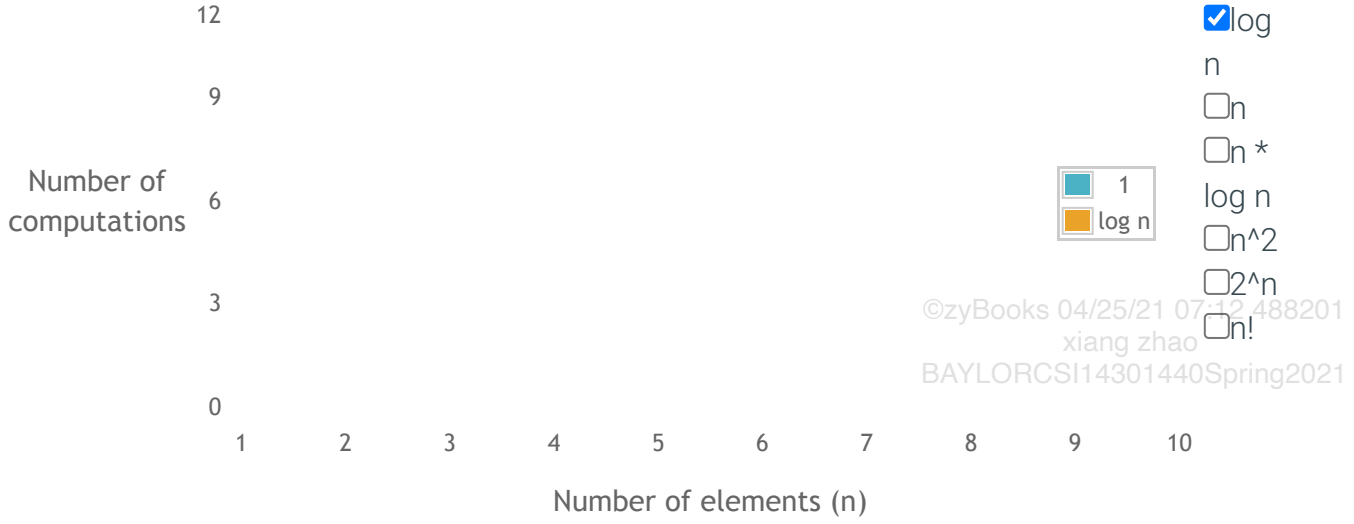
Function	N = 10	N = 50	N = 100	N = 1000	N = 10000	N = 100000
log N	3.3 µs	5.65 µs	6.6 µs	9.9 µs	13.3 µs	16.6 µs
N	10 µs	50 µs	100 µs	1000 µs	10 ms	0.1 s
N log N	.03 ms	.28 ms	.66 ms	.099 s	.132 s	1.66 s
N^2	.1 ms	2.5 ms	10 ms	1 s	100 s	2.7 hours
N^3	1 ms	.125 s	1 s	16.7 min	11.57 days	31.7 years
2^N	.001 s	35.7 years	*	*	*	*

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

PARTICIPATION ACTIVITY 9.2.4: Computational complexity graphing tool.

Number of computations vs number of elements

1



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 9.2.2: Runtime complexities for various pseudocode examples.

Notation	Name	Example pseudocode
O(1)	Constant	`FindMin(x, y) { if (x < y) { return x } else { return y } }`
O(log N)	Logarithmic	

		`` BinarySearch(numbers, N, key) { mid = 0; low = 0; high = 0; high = N - 1; while (high >= low) { mid = (high + low) / 2 if (numbers[mid] < key) { low = mid + 1 } else if (numbers[mid] > key) { high = mid - 1 } else { return mid } } return -1 // not found } ```
O(N)	Linear	``` LinearSearch(numbers, N, key) { for (i = 0; i < N; ++i) { if (numbers[i] == key) { return i } } return -1 // not found } ```
O(N log N)	Log-linear	``` MergeSort(numbers, i, k) { j = 0 if (i < k) { j = (i + k) / 2 // Find midpoint MergeSort(numbers, i, j) // Sort left part MergeSort(numbers, j + 1, k) // Sort right part Merge(numbers, i, j, k) // Merge parts } } ```
$O(N^2)$	Quadratic	``` SelectionSort(numbers, N) { for (i = 0; i < N; ++i) { indexSmallest = i for (j = i + 1; j < N; ++j) { if (numbers[j] < numbers[indexSmallest]) { indexSmallest = j } } temp = numbers[i] numbers[i] = numbers[indexSmallest] numbers[indexSmallest] = temp } ```
$O(c^N)$	Exponential	

```
Fibonacci(N) {
  if ((1 == N) || (2 == N)) {
    return 1
  }
  return Fibonacci(N-1) + Fibonacci(N-2)
}
```


PARTICIPATION ACTIVITY 9.2.5: 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)
   -

$O(N^2)$

9.3 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 9.3.1: Runtime analysis: Finding the max value.

Animation captions:

1. Runtime analysis determines the total number of operations. Operations include assignment, addition, comparison, etc.
2. The for loop iterates N times, but the for loop's initial expression i = 0 is executed once.
3. For each loop iteration, the increment and comparison expressions are each executed once. In the worst-case, the if's expression is true, resulting in 2 operations.
4. One additional comparison is made before the loop ends.
5. 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 9.3.2: Worst-case runtime analysis.

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


```
i = 0;
sum = 0;
while (i < N) {
   sum = sum + numbers[i];
   ++i;
}
```

- f(N) = 3N + 2
- f(N) = 3N + 3
- f(N) = 2 + N (N + 1)

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


```
negCount = 0;
for(i = 0; i < N; ++i) {
   if (numbers[i] < 0 ) {
      ++negCount;
   }
}
```

- f(N) = 2 + 4N + 1
- O(4N + 3)
- O(N)

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

```
for (i = 0; i < N; ++i) {
   if ((i % 2) == 0) {
      outVal[i] = inVals[i] * i;
   }
}
```

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

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

```
nVal = N;
steps = 0;
while (nVal > 0) {
   nVal = nVal / 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?

```
i = 0;
belowMinSum = 0.0;
belowMinCount = 0;
while (i < N && numbers[i] <=
maxVal) {
   belowMinCount = belowMinCount +
1;
   belowMinSum = numbers[i];
   ++i;
}
avgBelow = belowMinSum /
belowMinCount;
```


- 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 execute 5 operations before the loop, 3 operations 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 9.3.3: Simplified runtime analysis: A constant number of constant time operations is O(1).

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 9.3.4: Constant time operations.

1) A for loop of the form `for (i = 0; i < N; ++i) {}` that does not have nested loops or function calls, and does not modify i or N in the loop will always have a complexity of O(N).

True

False

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

```
if (timeHour < 6) {
   tollAmount = 1.55;
}
else if (timeHour < 10) {
   tollAmount = 4.65;
}
else if (timeHour < 18) {
   tollAmount = 2.35;
}
else {
   tollAmount = 1.55;
}
```


True

False

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

```
for (i = 0; i < 24; ++i) {
   if (timeHour < 6) {
      tollSchedule[i] = 1.55;
   }
   else if (timeHour < 10) {
      tollSchedule[i] = 4.65;
   }
   else if (timeHour < 18) {
      tollSchedule[i] = 2.35;
   }
   else {
      tollSchedule[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 9.3.5: Runtime analysis of nested loop: Selection sort algorithm.

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 9.3.1: Common summation: Summation of consecutive numbers.

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

PARTICIPATION ACTIVITY 9.3.6: Nested loops.

Determine the big-O worst-case runtime for each algorithm.

1)

```
for (i = 0; i < N; i++)  {
   for (j = 0; j < N; j++) {
      if (numbers[i] < numbers[j]) {
         ++eqPerms;
      }
      else {
         ++neqPerms;
      }
   }
}
```

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

2)

```
for (i = 0; i < N; i++)  {
   for (j = 0; j < (N - 1); j++) {
      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 = 0; i < N; i = i + 2)  {
   for (j = 0; j < N; j = j + 2) {
      cVals[i][j] = inVals[i] * j;
   }
}
```

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

4)

```
for (i = 0; i < N; ++i)  {
   for (j = i; j < N - 1; ++j) {
      cVals[i][j] = inVals[i] * j;
   }
}
```

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

5)

```
for (i = 0; i < N; ++i) {
   sum = 0;
   for (j = 0; j < N; ++j) {
      for (k = 0; k < N; ++k) {
         sum = sum + aVals[i][k] *
bVals[k][j];
      }

      cVals[i][j] = sum;
   }
}
```

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

9.4 Sorting: Introduction

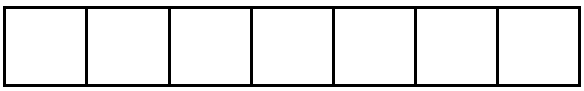
Sorting is the process of converting a list of elements into ascending (or descending) order. For example, given a list of numbers {17 3 44 6 9}, the list after sorting is {3 6 9 17 44}. You may have carried out sorting when arranging papers in alphabetical order, or arranging envelopes to have ascending 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.

PARTICIPATION ACTIVITY 9.4.1: Sort by swapping tool.

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

Start



Swap

Time - Best time -
Clear best

PARTICIPATION ACTIVITY 9.4.2: 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

9.5 Selection sort

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 9.5.1: 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; indexSmallest stores the index of the smallest element found.
3. Elements at i and indexSmallest 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 indexSmallest 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 9.5.2: 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 curly brackets in your answer.

Check Show answer

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

Figure 9.5.1: Selection sort algorithm.

```
#include <iostream>
using namespace std;

void SelectionSort(int numbers[], int numbersSize) {
   int i;
   int j;
   int indexSmallest;
   int temp;      // Temporary variable for swap

   for (i = 0; i < numbersSize - 1; ++i) {

      // Find index of smallest remaining element
      indexSmallest = i;
      for (j = i + 1; j < numbersSize; ++j) {

         if ( numbers[j] < numbers[indexSmallest] ) {
            indexSmallest = j;
         }
      }

      // Swap numbers[i] and numbers[indexSmallest]
      temp = numbers[i];
      numbers[i] = numbers[indexSmallest];
      numbers[indexSmallest] = temp;
   }
}

int main() {
   int numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 };
   const int NUMBERS_SIZE = 8;
   int i;

   cout << "UNSORTED: ";
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      cout << numbers[i] << ' ';
   }
   cout << endl;

   SelectionSort(numbers, NUMBERS_SIZE);

   cout << "SORTED: ";
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      cout << numbers[i] << ' ';
   }
   cout << endl;

   return 0;
}
```

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


©zyBooks 04/25/21 07:12 488201
xiang zhao
BAYLORCSI14301440Spring2021

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 9.5.3: Selection sort runtime.

1) When sorting a list with 50 elements, `indexSmallest` 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

CHALLENGE ACTIVITY 9.5.1: Selection sort.

Start

When using selection sort to sort a list with 16 elements, what is the minimum number of assignments to `indexSmallest`? Ex: 4

1 2 3 4

Check Next

9.6 Binary search

Linear search vs. 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 9.6.1: 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 9.6.2: 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 an array). 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 9.6.3: 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 9.6.1: Binary search algorithm.

```
BinarySearch(numbers, numbersSize, key) {
   mid = 0
   low = 0
   high = numbersSize - 1

   while (high >= low) {
      mid = (high + low) / 2
      if (numbers[mid] < key) {
         low = mid + 1
      }
      else if (numbers[mid] > key) {
         high = mid - 1
      }
      else {
         return mid
      }
   }

   return -1 // not found
}

main() {
   numbers = { 2, 4, 7, 10, 11, 32, 45, 87 }
   NUMBERS_SIZE = 8
   i = 0
   key = 0
   keyIndex = 0

   print("NUMBERS: ")
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      print(numbers[i] + " ")
   }
   printLine()

   print("Enter a value: ")
   key = getIntFromUser()

   keyIndex = BinarySearch(numbers, NUMBERS_SIZE, key)

   if (keyIndex == -1) {
      printLine(key + " was not found.")
   }
   else {
      printLine("Found " + key + " at index " + keyIndex + ".")
   }
}
```

```
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.
```


©zyBooks 04/25/21 07:12 488201
xiang zhao
BAYLORCSI14301440Spring2021

PARTICIPATION ACTIVITY 9.6.4: 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 an array 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 9.6.5: Speed of linear search versus binary search to find a number within a sorted list.

Animation captions:

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 9.6.6: Linear and binary search efficiency.

1) Suppose a list of 1024 elements is searched with linear search. How many distinct list elements are compared against a search key that is less than all elements in the list?

elements

Check Show answer

2) Suppose a sorted list of 1024 elements is searched with binary search. How many distinct list elements are compared against a search key that is less than all elements in the list?

elements

Check Show answer

CHALLENGE ACTIVITY 9.6.1: Binary search.

Start

A foods list is searched for Cream using binary search.
Foods list: (Bread, Butter, Cheese, Chocolate, Coffee, Cream, Milk, Oatmeal, Rice, Tea)

What is the first food searched?

Ex: Tea

What is the second food searched?

1 2 3 4 5

Check Next

9.7 Insertion sort

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 9.7.1: 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 9.7.1: Insertion sort algorithm.

```
#include <iostream>
using namespace std;

void InsertionSort(int numbers[], int numbersSize) {
   int i;
   int j;
   int temp;       // Temporary variable for swap

   for (i = 1; i < numbersSize; ++i) {
      j = i;
      // Insert numbers[i] into sorted part
      // stopping once numbers[i] in correct position
      while (j > 0 && 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;
      }
   }
}

int main() {
   int numbers[] = { 10, 2, 78, 4, 45, 32, 7, 11 };
   const int NUMBERS_SIZE = 8;
   int i;

   cout << "UNSORTED: ";
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      cout << numbers[i] << " ";
   }
   cout << endl;

   InsertionSort(numbers, NUMBERS_SIZE);

   cout << "SORTED: ";
   for(i = 0; i < NUMBERS_SIZE; ++i) {
      cout << numbers[i] << " ";
   }
   cout << endl;

   return 0;
}
```



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

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 9.7.2: 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 curly brackets in your answer.

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'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 9.7.3: 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

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 9.7.4: 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 9.7.5: 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 9.7.6: 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

9.8 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 9.8.1: 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 9.8.2: 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 is a faster algorithm for searching a list if the list's elements are sorted and directly accessible (such as an array). 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 9.8.3: Binary search efficiently searches sorted list by reducing the search

ACTIVITY

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 9.8.1: Binary search algorithm.

```
#include <iostream>
using namespace std;

int BinarySearch(int numbers[], int numbersSize, int key) {
   int mid;
   int low;
   int high;

   low = 0;
   high = numbersSize - 1;

   while (high >= low) {
      mid = (high + low) / 2;
      if (numbers[mid] < key) {
         low = mid + 1;
      }
      else if (numbers[mid] > key) {
         high = mid - 1;
      }
      else {
         return mid;
      }
   }

   return -1; // not found
}

int main() {
   int numbers[] = { 2, 4, 7, 10, 11, 32, 45, 87 };
   const int NUMBERS_SIZE = 8;
   int i;
   int key;
   int keyIndex;

   cout << "NUMBERS: ";
   for (i = 0; i < NUMBERS_SIZE; ++i) {
      cout << numbers[i] << ' ';
   }
   cout << endl;

   cout << "Enter a value: ";
   cin >> key;

   keyIndex = BinarySearch(numbers, NUMBERS_SIZE, key);

   if (keyIndex == -1) {
      cout << key << " was not found." << endl;
   }
   else {
      cout << "Found " << key << " at index " << keyIndex << "." << endl;
   }

   return 0;
}
```



```
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 9.8.4: Binary search algorithm execution.

Given sorted 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 an array 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 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 9.8.5: Speed of linear search versus binary search to find a number within a sorted list.

Animation captions:

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 9.8.6: Linear and binary search runtime.

Answer the following questions assuming each comparison takes 1 µs.

1) Given an unsorted 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 sorted list of 1024 elements, what is the runtime for binary search if the search key is greater than all elements in the list?

µs

Check Show answer