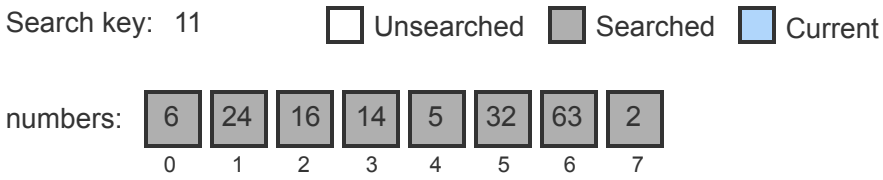
16.1 Searching and algorithms

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



Animation content:

Static figure:
Elements 6, 24, 16, 14, 5, 32, 63, and 2 are stored in a list at indices 0 to 7. When the search key is 11, all list elements are searched and 11 is not found in the list.

Step 1: Linear search starts at the first element and searches elements one-by-one.
Search key 14 is compared with elements 6, 24, 16 and 14 and is found at index 3.

Step 2: Linear search will compare all elements if the search key is not present.
Search key 11 is compared with all the elements in the list until the end of the list is reached.

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.

```
#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 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

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 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?

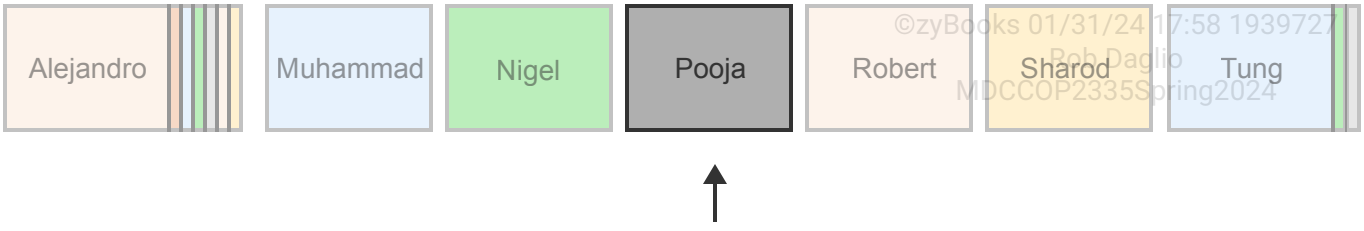
µs

Check Show answer

16.2 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 content:

Static figure:
There are various contact cards on the screen in a row, each of which has the name of a different contact, all of which are in alphabetical order. At the left side of the row, there is a stack of contact cards with only the top card's name visible, "Alejandro." Then, from left to right, the following names are in the same row as this stack of cards: "Muhammad," "Nigel," "Pooja"—which is highlighted and has an arrow pointing to it—"Robert," "Sharod," and "Tung"—under which is a stack of more contact cards, the names for which are not visible.

Step 1: A contact list stores contacts sorted by name. Searching for Pooja using a binary search starts by checking the middle contact. There is one stack of contact cards. Only the top card's name is visible, "Alejandro." The stack of contact cards then is split in half and the card that was in the middle is placed between these two stacks, resulting in, from left to right, one stack with the name "Alejandro" on top, a single card with the name "Muhammad," and then another stack with the name "Nigel" on top. An arrow appears that points to the card with the name "Muhammad."

Step 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. The arrow that was pointed at the card with the name "Muhammad" now points to the stack with "Nigel" as the top name.

Step 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. The arrow disappears. The stack with "Nigel" as the top name is split into two stacks and the card that was in the middle is placed between these two halves. The card in the middle has the name "Sharod." An arrow appears that points to this card that has the name "Sharod." This arrow then moves to the stack with the name "Nigel" at the top of the stack.

Step 4: The middle element between Muhammad and Sharod is Pooja. Each step reduces the number of contacts to search by half. The arrow disappears. The stack with "Nigel" as the top name is split into two stacks and the card that was in the middle is placed between these two halves. The card in the middle is "Pooja." An arrow appears that points to this card with the name "Pooja."

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 compared to?

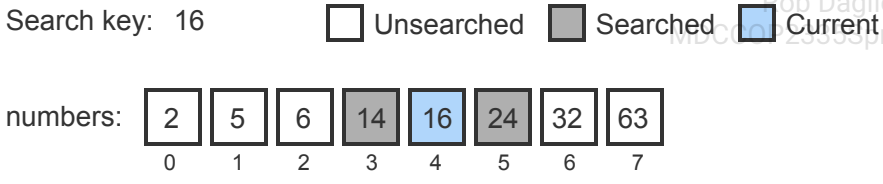
Check Show answer

2) What is the second contact compared to?

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



16 == 16

Search key found at index 4

Animation content:

Static figure:
There is an array with eight elements, each color-coded such that the index that stores the element is white if the index has not been searched, gray if the index has been searched, and blue if it is the index currently being searched. This array has integers as elements that are sorted in ascending order. The search key is "16." In the array, index 3, which has element "14," is gray; index 4, which has element "16," is blue; and index 5, which has element "24," is gray. Under this array is the text "16 == 16" and the text "Search key found at index 4."

Step 1: Elements with indices between low and high remain to be searched. Visible is the same array as in the static image, except all indices are colored white, meaning that all indices are unsearched. The text "low" appears below index 0 and the text "high" appears below index 7.

Step 2: Search starts by checking the middle element. The equation "mid = (low + high) / 2" appears. The values 0 and 7 take the places of "low" and "high" respectively and the expression evaluates to 3. The text "mid" then moves to rest underneath index 3 of the array and index 3 becomes colored blue. The inequality "16 != 14" then appears.

Step 3: If the search key is greater than element, then only elements in the right sublist need to be searched. The inequality "16 > 14" appears. The equation "low = mid + 1 = 4" appears and then the text "low" moves from being under index 0 to being under index 4. The equation "mid = (high + low)/2 = 5" appears and then the text "mid" moves from being under index 3 to being under index 5. Index 5 is colored blue, meaning index 5 is being searched, and then index 3 is colored gray, meaning that index 3 has been searched.

Step 4: Each iteration reduces search space by half. Search continues until the search key is found or the search space is empty. The inequality "16 < 24" appears. The equation "high = mid - 1 = 4" appears and then the text "high" moves from being under index 7 to being under index 4. The equation "mid = (4 + 4) /2 = 4" appears and then the text "mid" moves from being under index 5 to being under index 4. Index 4 is colored blue and index 5 is colored gray. The text "16 == 16" appears and then the text "Search key found at index 4" appears.

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.

©zyBooks 01/31/24 17:58 1939727
Rob Daglio
MDCCOP2335Spring2024

```
#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 16.2.4: 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

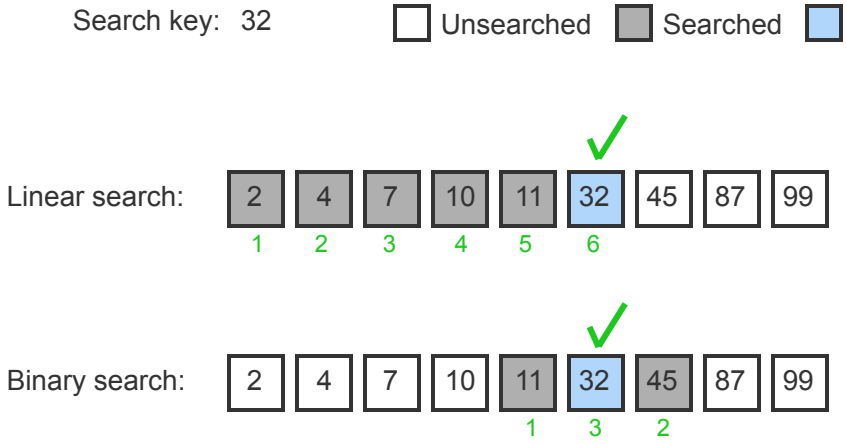
3) Given an array with 32 elements, how many list elements will be checked if the key is less than all elements in the list, using binary search?

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



Animation content:

Static figure:
Pictured are two arrays that contain the same integer elements in the same order, one labeled "Linear search" and the other labeled "Binary search." These are the elements in order: "2,4,7,10,11,32,45,87,99". Note that the arrays are sorted. The search key is "32." Each index is colored white if it is unsearched, gray if it has been searched, and blue if it is currently being searched. In the Linear-search array, indices 0 through and including 4 are gray and index 5 is blue; the rest of the indices are white. In the Binary-search array, indices 4 and 6 are gray and index 5 is blue; the rest of the indices are white. In both arrays there is a check mark over the index 5, which stores element 32. Additionally, the order in which each index was searched is given by a number below each index. In the Linear-search array, indices 0 through and including 5 are numbered 1 to 6 respectively. In the binary search array, indices 4, 5, and 6 and numbered 1, 3, 2 respectively.

Step 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.

Step 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.

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 16.2.6: 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

CHALLENGE ACTIVITY 16.2.1: Binary search.

539740.3879454.qx3zqy7

Start

A list of colors is searched for Green using binary search.
Colors list: { Beige Brick Brown Green Indigo Magenta Maroon Ochre Sienna Teal }

What is the first color searched?

Pick

What is the second color searched?

Pick

1 2 3 4 5

Check Next

16.3 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.

Algorithm steps: $5 + 13 \cdot N + 7 \cdot N^2$

Big O notation: $O(5 + 13 \cdot N + 7 \cdot N^2) = O(7 \cdot N^2)$

$= O(N^2)$ Pronounced: "Oh N squared"

Rule 1: If f(x) is a sum of several terms, the highest order term is kept and others are discarded.

Rule 2: If f(x) has a term that is a product of several factors, all constants are omitted.

Animation content:

Static figure:
Algorithm steps: 5 + 13·N + 7·N squared
Big O notation: O(5 + 13·N + 7·N squared)
= O(7·N squared)
= O(N squared) Pronounced: "Oh N squared"

Rule 1: If f(x) is a sum of several terms, the highest order term is kept and others are discarded.
Rule 2: If f(x) has a term that is a product of several factors, all constants are omitted.

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($734 \cdot N^2$)

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

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

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.

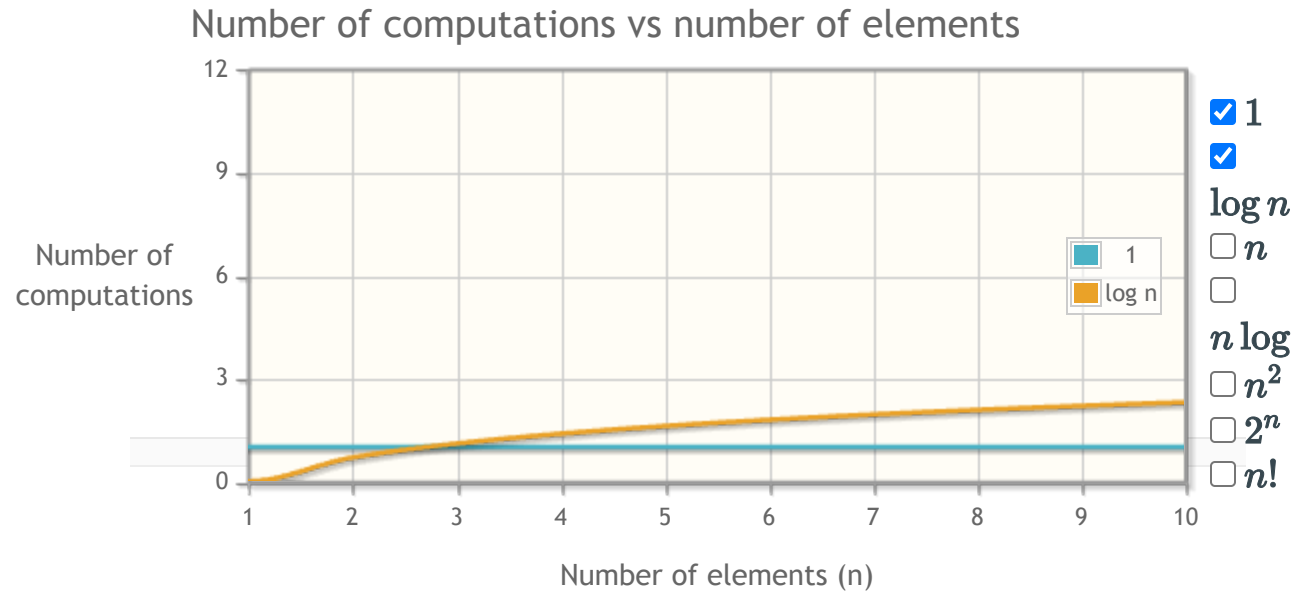
Table 16.3.1: Growth rates for different input sizes.

Function	N = 10	N = 50	N = 100	N = 1000	N = 10000	N = 100000
log 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 16.3.4: Computational complexity graphing tool.



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 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`
		`}`

©zyBooks 01/31/24 17:58 1939727
Rob Daglio
MDCCOP2335Spring2024

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 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.

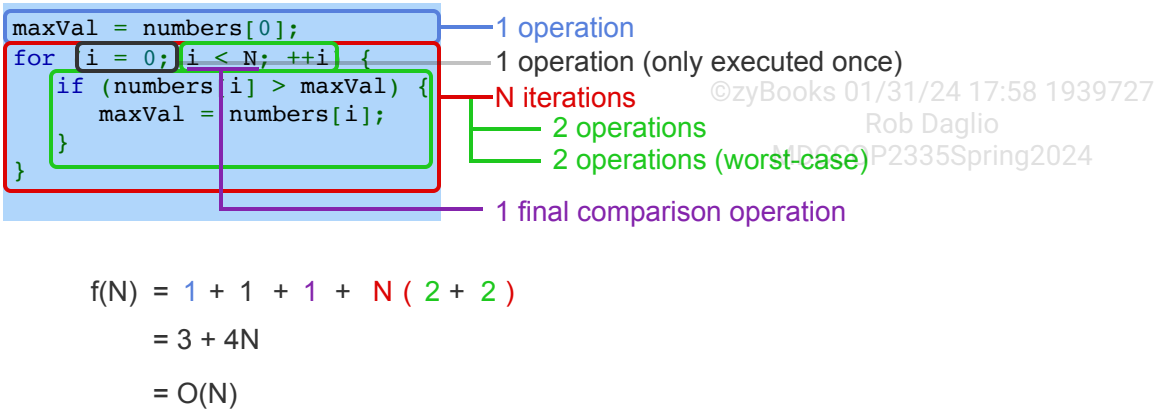
- O(N)
- O(N log N)
- $O(N^2)$

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.



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

Animation content:

Static figure: Code is shown with number of operations for each assignment, addition, comparison, etc. The function f(N) is evaluated as f(N)=1+1+1+N(2+2)=3+4N=O(N).
Begin code:

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

End code.
Step 1: Runtime analysis determines the total number of operations. Operations include assignment, addition, comparison, etc. The line of code, maxVal = numbers[0];, is boxed and represents 1 operation.
Step 2: The for loop iterates N times, but the for loop's initial expression i = 0 is executed once. The segment of code, i=0;, represents 1 operation. The for loop represents N iterations.
Step 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.
Step 4: One additional comparison, i < N, is made before the loop ends, resulting in 1 operation.
Step 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. f(N) is calculated by using the number of operations found at each step. 1 from the assignment operation plus 1 from the for loops initial expression operation plus 1 from the final comparison operation plus N for the for loop iterating N times and in parentheses 2 + 2 for when the if's expression is true resulting in 2 operations.

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 16.4.2: 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?

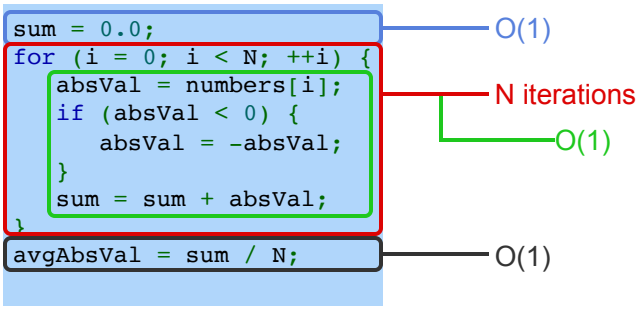
```
i = 0;
belowMinSum = 0.0;
belowMinCount = 0;
while (i < N && numbers[i] <=
maxVal) {
   belowMinCount =
belowMinCount + 1;
   belowMinSum = numbers[i];
   ++i;
}
avgBelow = belowMinSum /
belowMinCount;
```

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



O(1) + NO(1) + O(1) = O(N)

Animation content:

Static figure: Code is shown with number of constant time operations for each segment of code. Runtime analysis is evaluated as O(1)+NO(1)+O(1)= O(N)
Begin code:

```
sum = 0.0;
for (i = 0; i < N; ++i) {
  absVal = numbers[i];
  if (absVal < 0) {
    absVal = -absVal;
  }
  sum = sum + absVal;
}
avgAbsVal = sum / N;
```

End code:
Step 1: Constants are omitted in big-O notation, so any constant number of constant time operations is O(1). The line of code, sum = 0.0;, is O(1). The line of code inside the for loop are O(1). The line of code, sum = sum + absVal;, is O(1).

Step 2: The for loop iterates N times. Big-O complexity can be written as a composite function and simplified. Putting it all together results in O(1) from sum = 0.0 plus N times O(1) from the for loop and the inside of the for loop plus O(1) from avgAbsVal = sum / N which equals O(N) when simplified.

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 (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 16.4.5: Runtime analysis of nested loop: Selection sort algorithm.

```
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;
}
```

4 operations, generalized as constant c

Each of the N outer loop iterations executes 7 operations, generalized as constant d

$$f(N) = c\left((N - 1) + (N - 2) + \cdots + 2 + 1\right) = c\left(\frac{N(N-1)}{2}\right) + d \cdot N$$

$$O(f(N)) = O\left(\frac{c}{2}(N^2 - N) + d \cdot N\right) = O(N^2 + N) = O(N^2)$$

Animation content:

Static Figure:
Begin Java Code:

```
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;
}
```

End Java Code.

In the static figure, a box is around the code j < N; ++j . Another box is around the code
if (numbers[j] < numbers[indexSmallest]) {
 indexSmallest = j;
} . Text referring to these 2 boxes is displayed: 4 operations, generalized as constant c .

In the static figure, a box is around the code, indexSmallest = i; . A second box is around the code, j = i + 1; . A third box is around the code,
 temp = numbers[i];
 numbers[i] = numbers[indexSmallest];
 numbers[indexSmallest] = temp;
. Text referring to these 3 boxes is displayed: Each of the N outer loop iterations executes 5 operations, generalized as constant d.

In the static figure, the function f(N) is evaluated as:
f(N) = c((N-1) + (N-2) + ... + 2 + 1) = c(N(N-1)/2) + d·N

In the static figure, the function O(f(N)) is evaluated as:
O(f(N)) = O(c/2 (N^2 - N) + d·N) = O(N^2 + N) = O(N^2)

Step 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.
The line of code, for (i = 0; i < N; ++i) { , is highlighted. The text i = 0 is displayed.
The line of code, indexSmallest = i; is highlighted.
The line of code, for (j = i + 1; j < N; ++j) { is highlighted. The text j = 1 to N -1 is displayed. The text N - 1 iterations is displayed next to the line of code. The text f(N) = (N - 1) + is displayed.

Step 2: For i = 1, the inner loop iterates N - 2 times: iterating from j = 2 to N - 1.
The line of code, for (i = 0; i < N; ++i) { , is highlighted. The text i = 1 is displayed.
The line of code, indexSmallest = i; is highlighted.
The line of code, for (j = i + 1; j < N; ++j) { is highlighted. The text j = 2 to N - 1 is displayed. The text N - 2 iterations is displayed next to the line of code. The text f(N) = (N - 1) + (N - 2) + ... + is displayed.

Step 3: For i = N - 2, the inner loop iterates once: iterating from j = N - 1 to N - 1.
The line of code, for (i = 0; i < N; ++i) { , is highlighted. The text i = N - 2 is displayed.
The line of code, indexSmallest = i; is highlighted.
The line of code, for (j = i + 1; j < N; ++j) { is highlighted. The text j = N - 1 to N - 1 is displayed. The text 1 iteration is displayed next to the line of code. The text f(N) = (N - 1) + (N - 2) + ... + 2 + 1 is displayed.

Step 4: The summation is the sum of a consecutive sequence of numbers, and can be simplified.
The text = N(N - 1) / 2 is displayed next to the function.
The function f(N) now reads:
f(N) = (N - 1) + (N - 2) + ... + 2 + 1 = N(N-1)/2

Step 5: Each iteration of the loops requires a constant number of operations, which is defined as the constant c.
A box appears around the code j < N; ++j . A second box appears around the code
if (numbers[j] < numbers[indexSmallest]) {
 indexSmallest = j;
} . Text referring to these 2 boxes is displayed: 4 operations, generalized as constant c .
The function f(N) is updated to include the constant c. f(N) is now:
f(N) = c((N - 1) + (N - 2) + ... + 2 + 1) = c(N(N-1)/2)

Step 6: Additionally, each iteration of the outer loop requires a constant number of operations, which is defined as the constant d.
A box appears around the code in the outer for loop, i < N; ++i. A second box appears around the code, indexSmallest = i; . A third box appears around the code, j = i + 1; . A fourth box appears around the code,
 temp = numbers[i];
 numbers[i] = numbers[indexSmallest];
 numbers[indexSmallest] = temp;
. The following text referring to these 4 boxes is displayed: Each of the N outer loop iterations executes 7 operations, generalized as constant d.
The function f(N) is updated to add + d · N . f(N) is now:
f(N) = c((N - 1) + (N - 2) + ... + 2 + 1) = c(N(N-1)/2) + d · N

Step 7: Big-O notation omits the constant values, and the runtime is equal to the summation of the total inner loop iterations.

The Big O function equation O(f(N)) is displayed:
O(f(N)) = O(c/2 (N^2 - N) + d·N) = O(N^2 + N) = O(N^2)

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.

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)

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.

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".

Start

Swap

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

©zyBooks 01/31/24 17:58 1939727
Rob Daglio
MDCCOP2335Spring2024

16.6 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 16.6.1: Selection sort.


```
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;
}
```

Animation content:

Static figure:
Sorted list numbers has elements 3, 7, 8, 9, and 18 at indices 0 to 4.
Begin code:
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;
}
```

End code.

Step 1: Selection sort treats the input as two parts, a sorted and unsorted part. Variables i and j keep track of the two parts.
At the beginning the unsorted list has elements 7, 9, 3, 18, and 8 at indices 0 to 4.
The outer for loop starts from i = 0 and the inner for loop starts from j = 1.

Step 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.
First outer loop iteration: indexSmallest = 0.
Elements 7 and 9 are compared. 9 < 7 is false, so j = 2.
Elements 7 and 3 are compared. 3 < 7 is true, so indexSmallest =2 and j=3.
Elements 3 and 18 are compared. 18 < 3 is false, so j = 4.
Elements 3 and 8 are compared and 8 < 3 is false.

Step 3: Elements at i and indexSmallest are swapped.
The inner loop terminates. Variable temp is used to swap elements 7 and 3. List becomes: 3, 9, 7, 18, 8.

Step 4: Indices for the sorted and unsorted parts are updated.
Second outer loop iteration: i = 1, j = 2.

Step 5: The unsorted part is searched again, swapping the smallest element with the element at i.
indexSmallest = 1
Elements 9 and 7 are compared. 7 < 9 is true, so indexSmallest = 2 and j = 3.
Elements 7 and 18 are compared. 18 < 7 is false, so j = 4.
Elements 7 and 8 are compared and 8 < 7 is false.
The inner loop terminates and elements 9 and 7 are swapped. List becomes: 3, 7, 9, 18, 8.

Step 6: The process repeats until all elements are sorted.
Third outer loop iteration: i = 2, j =3
Elements 9 and 18 are compared. 18 < 9 is false, so j = 4.
Elements 9 and 8 are compared. 8 < 9 is true, so indexSmallest= 4.
The inner loop terminates and elements 9 and 8 are swapped. List becomes: 3, 7, 8, 18, 9.

Fourth outer loop iteration: i = 3, j =4 and indexSmallest = 3.

Elements 18 and 9 are compared. 9 < 18, so indexSmallest = 4.
The inner loop terminates and elements 18 and 9 are swapped. The sorted list is 3, 7, 8, 9, and 18.

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 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 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 16.6.1: 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;
}
```


©zyBooks 01/31/24 17:58 1939727
Rob Daglio
MDCCOP2335Spring2024

```
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, `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 16.6.1: Selection sort.

539740.3879454.qx3zqy7

Start

When using selection sort to sort a list with \(12\) elements, what is the minimum number of assignments to `indexSmallest`? Ex: 4

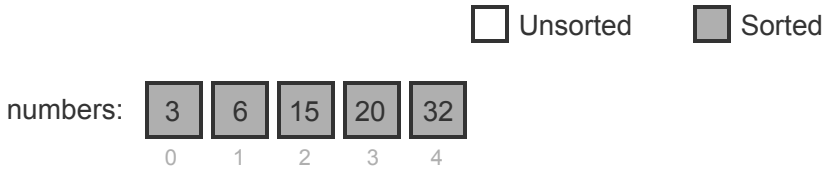
1 2 3 4

Check Next

16.7 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 16.7.1: Insertion sort.




```
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;
   }
}
```


Animation content:

Static figure:
Sorted list numbers has elements 3, 6, 15, 20, and 32 at indices 0 to 4.
Begin code:
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;
    }
}
```


End code.

Step 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. At the beginning, the unsorted list has elements 32, 6, 15, 3, and 20 at indices 0 to 4.
First outer loop iteration: i = j = 1.

Step 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.
Elements 32 and 6 are compared. 6 < 32 is true, so variable temp is used to swap elements 32 and 6. List becomes: 6, 32, 15, 3, 20 and j = 0, so the inner loop terminates.

Step 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.
Second outer loop iteration: i = j = 2.
Elements 32 and 15 are compared. 15 < 32 is true, so variable temp is used to swap elements 32 and 15. List becomes: 6, 15, 32, 3, 20 and j = 1.
Elements 6 and 15 are compared. 15 < 6 is false, so the inner loop terminates.

Step 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.
Third outer loop iteration: i = j = 3.
Elements 32 and 3 are compared. 3 < 32 is true, so variable temp is used to swap elements 32 and 3. List becomes: 6, 15, 3, 32, 20 and j = 2.
Elements 15 and 3 are compared. 3 < 15 is true, so variable temp is used to swap elements 15 and 3. List becomes: 6, 3, 15, 32, 20 and j = 1.
Elements 6 and 3 are compared. 3 < 6 is true, so variable temp is used to swap elements 6 and 3. List becomes: 3, 6, 15, 32, 20. j = 0, so the inner loop terminates.

Step 5: Once all elements in the unsorted part are inserted in the sorted part, the list is sorted.
Fourth outer loop iteration: i = j = 4.
Elements 32 and 20 are compared. 20 <32 is true, so variable temp is used to swap elements 32 and

20. List becomes: 3, 6, 15, 20, 32 and j = 3.
Elements 15 and 20 are compared. 20 < 15 is false, so j = 2.
Elements 6 and 20 are compared. 20 < 15 is false, so the inner loop terminates.
The sorted list is 3, 6, 15, 20, and 32.

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.

```
#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
```


©zyBooks 01/31/24 17:58 1939727
Rob Daglio
MDCCOP2335Spring2024

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 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 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

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}

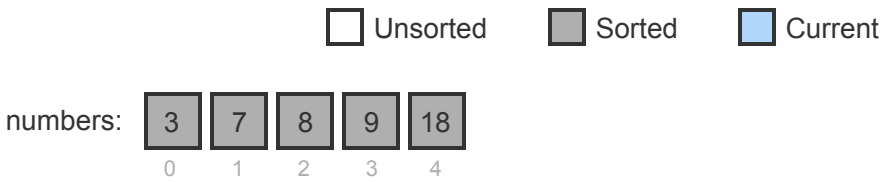
- Unsorted
- Sorted
- Nearly sorted

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 content:

Static figure:
Sorted list numbers has elements 3, 7, 8, 9, 18 at indices 0 to 4.
Step 1: Unsorted part initially contains the first element.
At the beginning, the nearly sorted list has elements 3, 7, 9, 18, 8 at indices 0 to 4.

Step 2: An element already in sorted position only requires a single comparison, which is O(1) complexity.
Elements 3 and 7, 7 and 9, and 9 and 18 are compared.

Step 3: An element not in sorted position requires O(N) comparisons. For nearly sorted inputs, insertion sort's runtime is O(N).
Elements 18 and 8 are compared, 18 > 8 is true, so 18 and 8 are swapped. List becomes: 3, 7, 9, 8,

18.
Elements 9 and 8 are compared, 9 > 8 is true, so 9 and 8 are swapped. List becomes: 3, 7, 8, 9, 18.
Elements 7 and 8 are compared, 7 > 8 is false, so the sorting stops.

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 15}, 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

CHALLENGE ACTIVITY 16.7.1: Insertion sort.

539740.3879454.qx3zqy7

Start

Given list {22 23 29 30 31 34 36 38 28}, what is the value of i when the first swap executes?

Ex: 1

1	2	3	4	5

Check Next

16.8 Quicksort

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.

Low partition
High partition

midpoint = i + (k - i) / 2
= 0 + (4 - 0) / 2

= 2

pivot: 6

```
int Partition(int numbers[], int i, int k) {
   /* Pick middle value as pivot */
   midpoint = i + (k - i) / 2;
   pivot = numbers[midpoint];

   /* Initialize variables */
   l = i;
   h = k;

   while (!done) {
      /* Increment l while numbers[l] < pivot */
      while (numbers[l] < pivot) {
         ++l;
      }

      /* Decrement h while pivot < numbers[h] */
      while (pivot < numbers[h]) {
         --h;
      }
```



```
      /* 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;
         --h;
      }
   }
   return h;
}
```

Animation content:

Static figure:
Unsorted list numbers has elements 6, 4, 7, 18 and 8 at indices 0 to 4.
Elements 6 and 4 are in the low partition, and elements 7, 18 and 8 are in the high partition.
midpoint = i + (k - i) / 2 = 0 + (4 - 0) / 2 = 2
pivot: 6
Begin code:

```
int Partition(int numbers[], int i, int k) {
  /* Pick middle value as pivot */
  midpoint = i + (k - i) / 2;
  pivot = numbers[midpoint];

  /* Initialize variables */
  l = i;
  h = k;

  while (!done) {
    /* Increment l while numbers[l] < pivot */
    while (numbers[l] < pivot) {
      ++l;
    }
```

```
    /* Decrement h while pivot < numbers[h] */
    while (pivot < numbers[h]) {
      --h;
    }

    /* 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;
      --h;
    }
  }
  return h;
}
```


End code.

Step 1: The pivot value is the value of the middle element.
Initially, i = 0 and k = 4.
midpoint = i + (k - i) / 2 = 0 + (4 - 0) / 2 = 2, so pivot = numbers[2]= 6

Step 2: Index l begins at element i and is incremented until a value greater than the pivot is found.
l = i = 0.
Elements 7 and 6 are compared, 7 < 6 is false, so l remains unchanged.

Step 3: Index h begins at element k, and is decremented until a value less than the pivot is found.
h = k = 4.
Elements 6 and 8 are compared, 6 < 8 is true, so h = 3.
Elements 6 and 18 are compared, 6 < 18 is true, so h = 2.
Elements 6 and 6 are compared, 6 < 6 is false, so h remains unchanged.

Step 4: Elements at indices l and h are swapped, moving those elements to the correct partitions.
l and h are compared. 0 >=2 is false, so variable temp is used to swap elements 7 and 6.
List becomes: 6, 4, 7, 18, 8.

Step 5: The partition process repeats until indices l and h reach or pass each other (l >= h), indicating all elements have been partitioned.
l = 1 and h = 1.
Elements 4 and 6 are compared. 4 < 6 is true, so l = 2.
Elements 7 and 6 are compared. 7 < 6 is false, so l remains unchanged.
Elements 6 and 4 are compared, 6 < 4 is false, so h remains unchanged.
l and h are compared. 2 >= 1 is true, so done = true.

Step 6: Once partitioned, the algorithm returns h indicating the highest index of the low partition. The partitions are not yet sorted.
h = 1

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

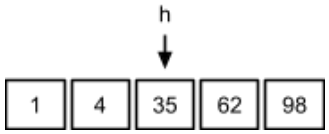
pivot =

Check Show answer

PARTICIPATION ACTIVITY 16.8.3: Low and high partitions.

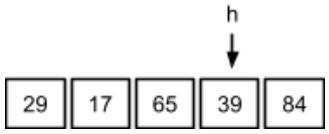
Determine if the low and high partitions are correct given h and pivot.

1) pivot = 35



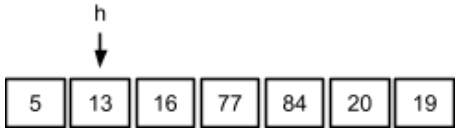
- True
- False

2) pivot = 65



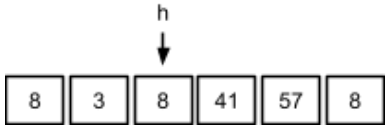
- True
- False

3) pivot = 5



- True
- False

4) pivot = 8

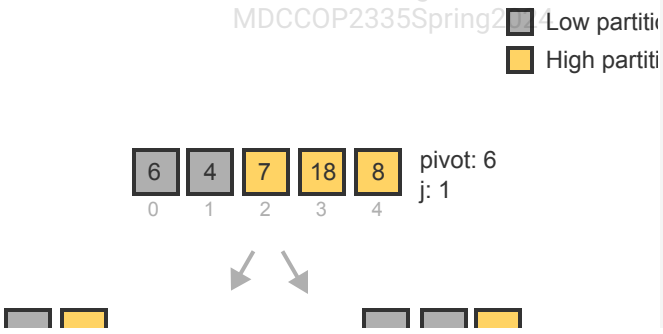


- True
- False

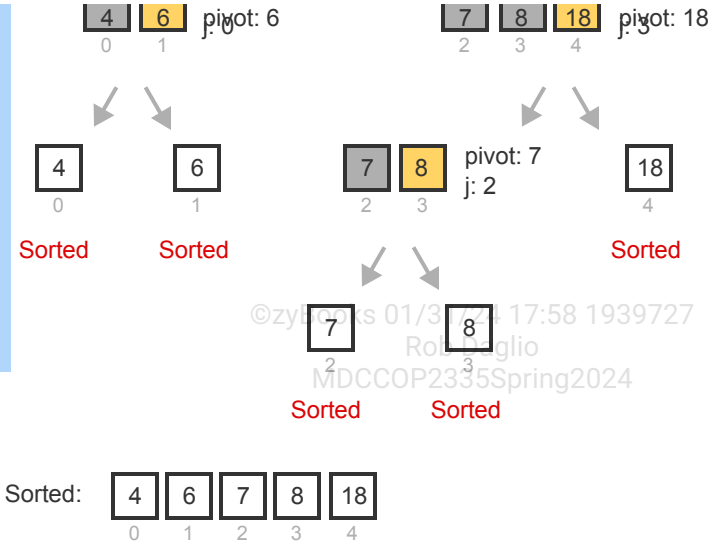
Once partitioned, each partition needs to be sorted. Quicksort is typically implemented as a recursive algorithm using calls to quicksort to sort the low and high partitions. This recursive sorting process continues until a partition has one or zero elements, and thus already sorted.

PARTICIPATION ACTIVITY 16.8.4: Quicksort.


```
void Quicksort(int numbers[], int i, int k) {
   int j;

   /* Base case: If 1 or zero elements,
      partition is already sorted */
   if (i >= k) {
      return;
```



```
   }

   /* Partition the array.
      Value j is the location of last
      element in low partition. */
   j = Partition(numbers, i, k);

   /* Recursively sort low and high
       partitions */
   Quicksort(numbers, i, j);
   Quicksort(numbers, j + 1, k);
}
```



Animation content:

Static figure:
Unsorted list with elements 6, 4, 7, 18 and 8 at indices 0 to 4 is partitioned with pivot: 6 and j: 1.
Quicksort is called recursively on the low and high partitions.
Low partition 4, 6 at indices 0 and 1. High partition 7, 8, 18 at indices 2 to 4.
Low partition 4 at index 0. High partition 6 at index 1. Partitions 4 and 6 are sorted.
Low partition 7, 8 at indices 2, 3. High partition 18 at index 4. Partition 18 is sorted.
Low partition 7 at index 2. High partition 8 at index 3. Partitions 7 and 8 are sorted.
Sorted list numbers contains elements 4, 6, 7, 8, and 18 at indices 0 to 4.

Begin code:

```
void Quicksort(int numbers[], int i, int k) {
  int j;

  /* Base case: If 1 or zero elements,
     partition is already sorted */
  if (i >= k) {
     return;
  }

  /* Partition the array.
     Value j is the location of last
     element in low partition. */
  j = Partition(numbers, i, k);

  /* Recursively sort low and high
      partitions */
  Quicksort(numbers, i, j);
```

```
    Quicksort(numbers, j + 1, k);
}
End code.
```

Step 1: List contains more than one element. Partition the list.
Initially, the list has elements 7, 4, 6, 18 and 8 at indices 0 to 4.
The list is partitioned with pivot: 6 and j: 1. List becomes: 6, 4, 7, 18, 8.

Step 2: Recursively call quicksort on the low and high partitions.
Quicksort is called on low partition 6,4 at indices 0 and 1, and on high partition 7, 18, 8 at indices 2 to 4.

Step 3: Low partition contains more than one element. Partition the low partition and recursively call quicksort.
Low partition 6, 4 is partitioned, so pivot: 6 and j:0, and elements 6 and 4 are swapped.
Quicksort is called on low partition 4 at index 0, and high partition 6 at index 1.

Step 4: Low partition contains one element, so partition is already sorted. High partition contains one element, so partition is already sorted.
Low partition 4 and high partition 6 are sorted.

Step 5: High partition contains more than one element. Partition the high partition and recursively call quicksort.
High partition 7, 18, 8 is partitioned, so pivot: 18 and j:3, and elements 18 and 8 are swapped.
Quicksort is called on low partition 7, 8 at indices 2, 3, and high partition 18 at index 4.

Step 6: Low partition contains more than one element. Partition the low partition and recursively call quicksort.
Low partition 7, 8 is partitioned, so pivot: 7 and j: 2.
Quicksort is called on low partition 7 at index 2, and high partition 8 at index 3.

Step 7: Low partition contains one element, so partition is already sorted. High partition contains one element, so partition is already sorted.
Low partition 7 and high partition 8 are sorted.

Step 8: High partition contains one element, so partition is already sorted.
High partition 18 is sorted.

Step 9: All elements are sorted.
Sorted list numbers contains elements 4, 6, 7, 8, and 18 at indices 0 to 4.

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.

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

Figure 16.8.1: Quicksort algorithm.

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


int Partition(int numbers[], int i, int k) {
   int l;
   int h;
   int midpoint;
   int pivot;
   int temp;
   bool done;

   /* Pick middle element as pivot */
   midpoint = i + (k - i) / 2;
   pivot = numbers[midpoint];

   done = false;
   l = i;
   h = k;

   while (!done) {

      /* Increment l while numbers[l] < pivot */
      while (numbers[l] < pivot) {
         ++l;
      }

      /* Decrement h while pivot < numbers[h] */
      while (pivot < numbers[h]) {
         --h;
      }

      /* If there are zero or one elements 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;
         --h;
      }
   }

   return h;
}

void Quicksort(int numbers[], int i, int k) {
   int j;

   /* Base case: If there are 1 or zero elements 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 element 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);
}

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;

   /* Initial call to quicksort */
   Quicksort(numbers, 0, NUMBERS_SIZE - 1);

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

   return 0;
}
```

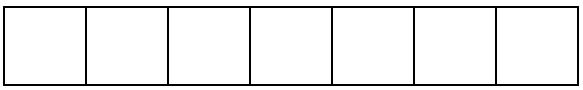
```
UNSORTED: 10 2 78 4 45 32 7 11
SORTED: 2 4 7 10 11 32 45 78
```


The 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".

Start



Partition Back

Time - Best time -
Clear best

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 are the contents of the low partition? Use curly braces in your answer.

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

CHALLENGE ACTIVITY 16.8.1: Quicksort.

539740.3879454.qx3zqy7

Start

Given numbers = {33 25 90 37 60}, i = 0, k = 4

What is the midpoint? Ex: 9

What is the pivot?

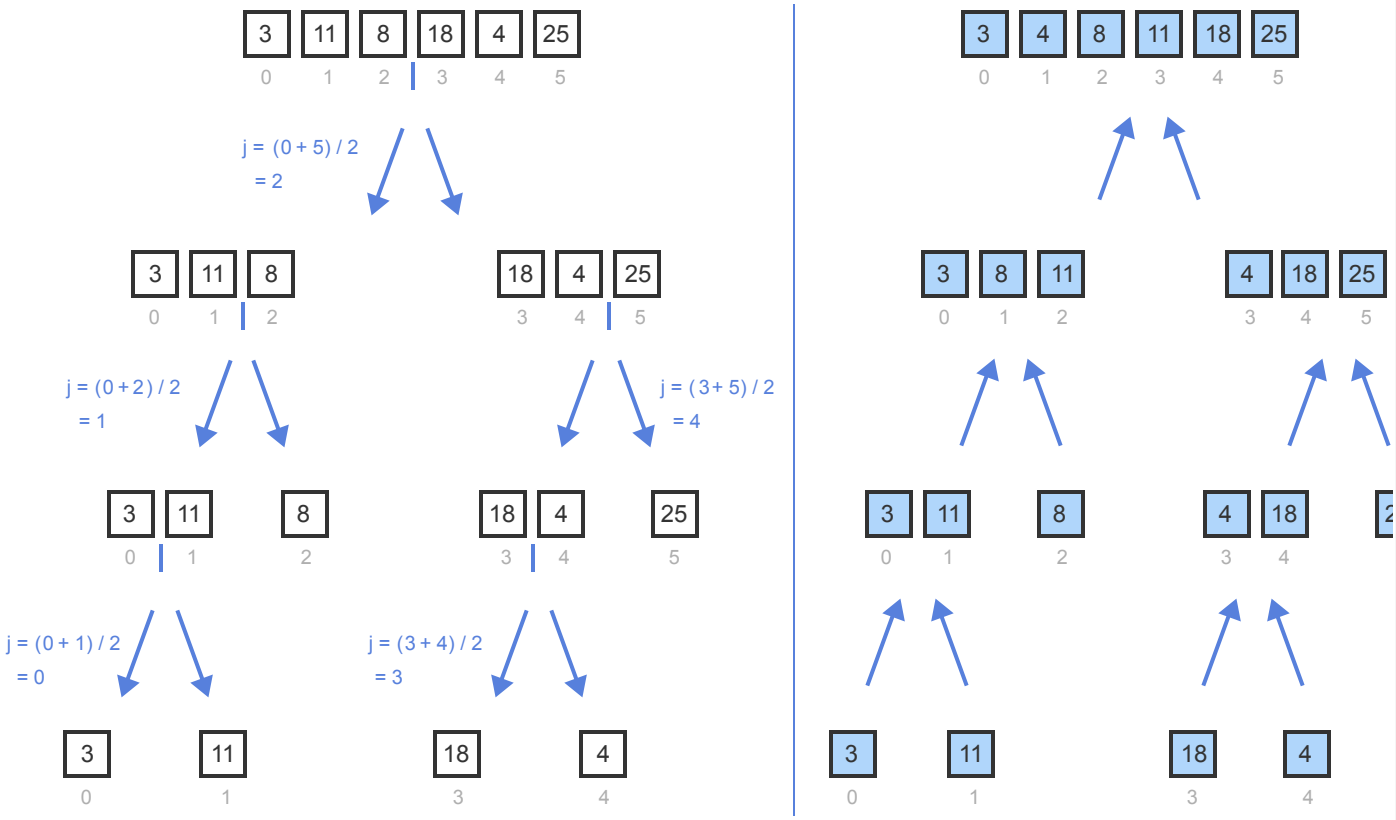
1 2 3 4 5 6

Check Next

16.9 Merge sort

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 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:

Static figure:
Unsorted list has elements 3, 11, 8, 18, 4, and 25 at indices 0 to 5.
List is divided with j = (0 + 5) / 2 = 2. First half: 3, 11, 8 at indices 0 to 2. Second half: 18, 4, 25 at indices 3 to 5.

First half is divided with j = (0 + 2) / 2 = 1 to 3, 11 at indices 0, and 1 and 8 at index 3.
3, 11 is divided with j = (0 + 1) / 2 = 0 to 3 at index 0, and 11 at index 1.
Second half is divided with j = (3 + 5) / 2 = 4 to 18, 44 at indices 3 and 4, and 25 at index 5.

18, 4 is divided with j = (3 + 4) / 2 = 3 to 18 at index 3, and 4 at index 4.

3 and 11 are merged to 3, 11.
3, 11 and 8 are merged to 3, 8, 11.
18 and 4 are merged to 4, 18.
4, 18 and 25 are merged to 4, 18, 25.
3, 8, 11 and 4, 18, 25 are merged and the final list becomes: 3, 4, 8, 11, 18, 25

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 function call. The index variable i is the index of 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

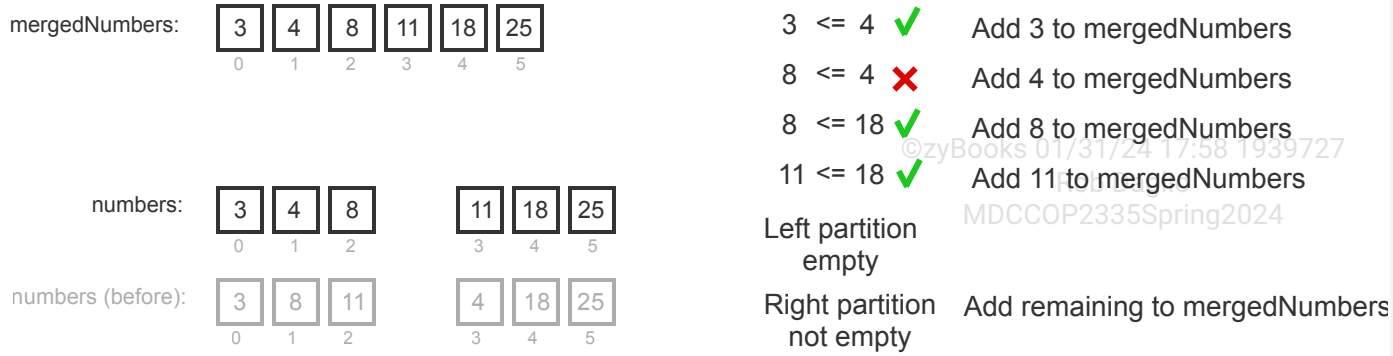
6) numbers = {34 78 14 23 8 35}, i = 3, k = 5

```
Right partition  = {
}
```

Check Show answer

Merge sort 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 left or right partition is added one at a time to a temporary merged list. Once merged, temporary list is copied back to the original list.




```
void Merge(int numbers[], int i, int j, int k) {
```

```
// If left partition not empty, add remaining elements
   while (leftPos <= j) {
      mergedNumbers[mergePos] = numbers[leftPos];
```

```
   // Create temporary array mergedNumbers
   // Initialize position variables

   // Add smallest element to merged numbers
   while (leftPos <= j && rightPos <= k) {
      if (numbers[leftPos] <= numbers[rightPos]) {
         mergedNumbers[mergePos] = numbers[leftPos];
         ++leftPos;
      }
      else {
         mergedNumbers[mergePos] = numbers[rightPos];
         ++rightPos;
      }
      ++mergePos;
   }
      ++leftPos;
      ++mergePos;
   }

   // If right partition not empty, add remaining elements
   while (rightPos <= k) {
      mergedNumbers[mergePos] = numbers[rightPos];
      ++rightPos;
      ++mergePos;
   }

   // Copy merge number back to numbers
   for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
      numbers[i + mergePos] = mergedNumbers[mergePos];
   }
}
```


Animation content:

Static figure:
Begin code:

```
void Merge(int numbers[], int i, int j, int k) {
  // Create temporary array mergedNumbers
  // Initialize position variables

  // Add smallest element to merged numbers
  while (leftPos <= j && rightPos <= k) {
    if (numbers[leftPos] <= numbers[rightPos]) {
      mergedNumbers[mergePos] = numbers[leftPos];
      ++leftPos;
    }
    else {
      mergedNumbers[mergePos] = numbers[rightPos];
      ++rightPos;
    }
    ++mergePos;
  }

 // If left partition not empty, add remaining elements
  while (leftPos <= j) {
    mergedNumbers[mergePos] = numbers[leftPos];
    ++leftPos;
    ++mergePos;
  }

  // If right partition not empty, add remaining elements
  while (rightPos <= k) {
    mergedNumbers[mergePos] = numbers[rightPos];
    ++rightPos;
```

```
      ++mergePos;
    }

    // Copy merge number back to numbers
    for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
      numbers[i + mergePos] = mergedNumbers[mergePos];
    }
  }
End code.
```


List numbers initially has elements 3, 8, 11, 4, 18, and 25 at indices 0 to 5.
3 is compared with 4. 3 <= 4 is true, so 3 is added to mergedNumbers.
8 is compared with 4. 8 <= 4 is false, so 4 is added to mergedNumbers.
8 is compared with 18. 8 <= 18 is true, so 8 is added to mergedNumbers.
11 is compared with 18. 11 <= 18 is true, so 11 is added to mergedNumbers.

The left partition is empty and the right partition is not empty, so the right partition's remaining elements are added to mergedNumbers.

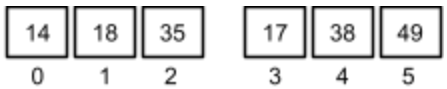
Lists numbers and mergedNumbers both have elements 3, 4, 8, 11, 18 and 25 at indices 0 to 5.

Animation captions:

1. Create a temporary list for merged numbers. Initialize mergePos, leftPos, and rightPos 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. Copy the elements in the temporary list back to the original list.

PARTICIPATION ACTIVITY 16.9.4: Tracing merge operation.

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



1) leftPos = 0, rightPos = 3

Check Show answer

2) leftPos = 1, rightPos = 3

Check Show answer

3) leftPos = 1, rightPos = 4

Check Show answer

4) leftPos = 2, rightPos = 4

Check Show answer

5) leftPos = 3, rightPos = 4

Check Show answer

6) leftPos = 3, rightPos = 5

Check Show answer

Figure 16.9.1: Merge sort algorithm.

©zyBooks 01/31/24 17:58 1939727
Rob Daglio
MDCCOP2335Spring2024

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

void Merge(int numbers[], int i, int j, int k) {
   int mergedSize;                              // Size of merged partition
   int mergePos;                                // Position to insert merged
number
   int leftPos;                                 // Position of elements in
left partition
   int rightPos;                                // Position of elements in
right partition
   int* mergedNumbers = nullptr;

   mergePos = 0;
   mergedSize = k - i + 1;
   leftPos = i;                                 // Initialize left partition
position
   rightPos = j + 1;                            // Initialize right partition
position
   mergedNumbers = new int[mergedSize];         // Dynamically allocates
temporary array
                                                // for merged numbers

   // Add smallest element from left or right partition to merged numbers
   while (leftPos <= j && rightPos <= k) {
      if (numbers[leftPos] < numbers[rightPos]) {
         mergedNumbers[mergePos] = numbers[leftPos];
         ++leftPos;
      }
      else {
         mergedNumbers[mergePos] = numbers[rightPos];
         ++rightPos;

      }
      ++mergePos;
   }

   // If left partition is not empty, add remaining elements to merged
numbers
   while (leftPos <= j) {
      mergedNumbers[mergePos] = numbers[leftPos];
      ++leftPos;
      ++mergePos;
   }

   // If right partition is not empty, add remaining elements to merged
numbers
   while (rightPos <= k) {
      mergedNumbers[mergePos] = numbers[rightPos];
      ++rightPos;
      ++mergePos;
   }

   // Copy merge number back to numbers
   for (mergePos = 0; mergePos < mergedSize; ++mergePos) {
      numbers[i + mergePos] = mergedNumbers[mergePos];
   }
   delete[] mergedNumbers;
}
```

```
void MergeSort(int numbers[], int i, int k) {
   int j;

   if (i < k) {
      j = (i + k) / 2;  // Find the midpoint in the partition

      // Recursively sort left and right partitions
      MergeSort(numbers, i, j);
      MergeSort(numbers, j + 1, k);

      // Merge left and right partition in sorted order
      Merge(numbers, i, j, k);
   }
}

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;

   MergeSort(numbers, 0, NUMBERS_SIZE - 1);

   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 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.

To allocate the temporary array, the `Merge()` function dynamically allocates the array. mergedNumbers is a pointer variable that points to the dynamically allocated array, and

`new int[mergedSize]` allocates the array with mergedSize elements. Alternatively, instead of allocating the array within the `Merge()` function, a temporary array with the same size as the array being sorted can be passed as an argument.

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

CHALLENGE ACTIVITY 16.9.1: Merge sort.

539740.3879454.qx3zqy7

Start

numbers: | 77 | 39 | 18 | 50 | 16 | 36 |

What call sorts the numbers array?

MergeSort(numbers, Ex: 1 ,)

1 2 3 4 5

Check Next

16.10 LAB: Descending selection sort with output during execution

Write a void function SelectionSortDescendTrace() that takes an integer array and sorts the array into descending order. The function should use nested loops and output the array after each iteration of the outer loop, thus outputting the array N-1 times (where N is the size). Complete main() to read in a list of up to 10 positive integers (ending in -1) and then call the SelectionSortDescendTrace() function. For coding simplicity, output a space after every integer, including the last one on each row.

If the input is:

```
20 10 30 40 -1
```

then the output is:

```
40 10 30 20
40 30 10 20
40 30 20 10
```


539740.3879454.qx3zqy7

LAB ACTIVITY 16.10.1: LAB: Descending selection sort with output during execution 0 / 10

main.cpp Load default template...

```
1 Loading latest submission...
```


Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program

Input (from above) → **main.cpp** (Your program) → Outp

Program output displayed here

Coding trail of your work What is this?

History of your effort will appear here once you begin working on this zyLab.

16.11 LAB: Sorting user IDs

Given a main() that reads user IDs (until -1), complete the Quicksort() and Partition() functions to sort the IDs in ascending order using the Quicksort algorithm, and output the sorted IDs one per line.

Ex. If the input is:


```
kaylasimms
julia
myron1994
kaylajones
-1
```

the output is:

```
julia
kaylajones
kaylasimms
myron1994
```


LAB ACTIVITY | 16.11.1: LAB: Sorting user IDs | 0 / 10

main.cpp — Load default template...

```
1 Loading latest submission...
```


Develop mode | Submit mode

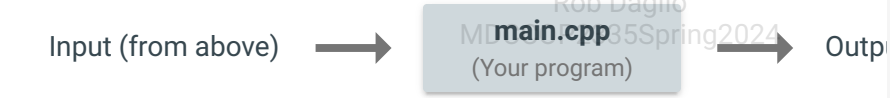
Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first

box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

16.12 LAB: Insertion sort

The program has four steps:

1. Read the size of an integer array, followed by the elements of the array (no duplicates).
2. Output the array.
3. Perform an insertion sort on the array.
4. Output the number of comparisons and swaps performed.

main() performs steps 1 and 2.

Implement step 3 based on the insertion sort algorithm in the book. Modify InsertionSort() to:

- Count the number of comparisons performed.
- Count the number of swaps performed.
- Output the array during each iteration of the outside loop.

Complete main() to perform step 4, according to the format shown in the example below.

Hints: In order to count comparisons and swaps, modify the while loop in InsertionSort(). Use global variables for comparisons and swaps.

The program provides three helper functions:

```
// Read size numbers from cin into a new array and return the array.
int* ReadNums(int size)

// Print the numbers in the array, separated by spaces
// (No space or newline before the first number or after the last.)
void PrintNums(int nums[], int size)

// Exchange nums[j] and nums[k].
void Swap(int nums[], int j, int k)
```


Ex: When the input is:

```
6 3 2 1 5 9 8
```

the output is:

```
 3 2 1 5 9 8

 2 3 1 5 9 8
 1 2 3 5 9 8
 1 2 3 5 9 8
 1 2 3 5 9 8
 1 2 3 5 8 9

comparisons: 7
swaps: 4
```


LAB ACTIVITY | 16.12.1: LAB: Insertion sort | 0 / 10

main.cpp — Load default template...

```
1 Loading latest submission...
```

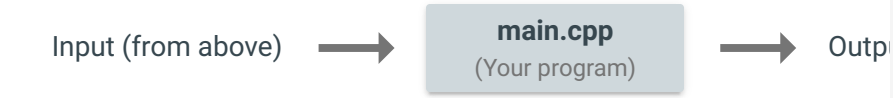
Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

16.13 LAB: Merge sort

The program is the same as shown at the end of the Merge sort section, with the following changes:

- Numbers are entered by a user in a separate helper function, ReadNums(), instead of defining a specific array in main(). The first number is how many integers to be sorted, and the rest are the integers.
- Output of the array has been moved to the helper function PrintNums().
- An output has been added to MergeSort(), showing the indices that will be passed to the recursive function calls.

Add code to the merge sort algorithm to count the number of comparisons performed.

Add code at the end of main() that outputs "comparisons: " followed by the number of comparisons performed (Ex: "comparisons: 12")

Hint: Use a global variable to count the comparisons.

Note: Take special care to look at the output of each test to better understand the merge sort algorithm.

Ex: When the input is:


```
6 3 2 1 5 9 8
```

the output is:

```
unsorted: 3 2 1 5 9 8

0 2 | 3 5
0 1 | 2 2
0 0 | 1 1
3 4 | 5 5
3 3 | 4 4

sorted:   1 2 3 5 8 9
comparisons: 8
```


LAB ACTIVITY | 16.13.1: LAB: Merge sort | 0 / 10

main.cpp — Load default template...

```
1 Loading latest submission...
```

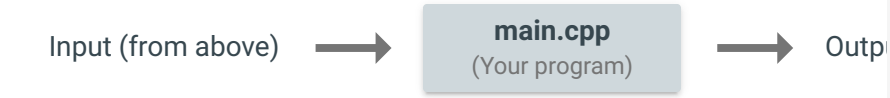
Develop mode | Submit mode

Run your program as often as you'd like, before submitting for grading. Below, type any needed input values in the first box, then click **Run program** and observe the program's output in the second box.

Enter program input (optional)

If your code requires input values, provide them here.

Run program



Program output displayed here

Coding trail of your work What is this?

```
History of your effort will appear here once you begin working
on this zyLab.
```

16.14 LAB: Binary search

Binary search can be implemented as a recursive algorithm. Each call makes a recursive call on one-half of the list the call received as an argument.

Complete the recursive function BinarySearch() with the following specifications:

1. Parameters:
   - a target integer
   - a vector of integers
   - lower and upper bounds within which the recursive call will search
2. Return value:
   - the index within the vector where the target is located
   - -1 if target is not found

The template provides the main program and a helper function that reads a vector from input.

The algorithm begins by choosing an index midway between the lower and upper bounds.

1. If `target == integers.at(index)` return index
2. If `lower == upper`, return -1 to indicate not found
3. Otherwise call the function recursively on half the vector parameter:
    - If `integers.at(index) < target`, search the vector from index + 1 to upper
    - If `integers.at(index) > target`, search the vector from lower to index - 1

The vector must be ordered, but duplicates are allowed.

Once the search algorithm works correctly, add the following to BinarySearch():

4. Count the number of calls to BinarySearch().
5. Count the number of times when the target is compared to an element of the vector. Note: lower == upper should not be counted.

Hint: Use a global variable to count calls and comparisons.

The input of the program consists of:

1. the number of integers in the vector
2. the integers in the vector
3. the target to be located

Ex: If the input is:

```
9
1 2 3 4 5 6 7 8 9
2
```

the output is:

```
index: 1, recursions: 2, comparisons: 3
```


LAB ACTIVITY | 16.14.1: LAB: Binary search | 0 / 10

main.cpp — Load default template...

```
1 Loading latest submission...
```