Programming Refresher Workshop

Session 3 A/P Tan Sun Teck

Contents

1. Motivation #1: Coin Change (1/2)

- Some of the programs we have written are "long-winded", because we have not learned enough C constructs to make it simpler.
- Consider the Coin Change problem with 6 denominations 1¢, 5¢, 10¢, 20¢, 50¢, and $1:

```
Algorithm:
    input: amt (in cents); output: coins
    coins ← 0
    coins += amt/100; amt %= 100;
    coins += amt/50;   amt %= 50;
    coins += amt/20;   amt %= 20;
    coins += amt/10;   amt %= 10;
    coins += amt/5;    amt %= 5;
    coins += amt/1;    amt %= 1;
    print coins
```

1. Motivation #1: Coin Change (2/2)

Week7_CoinChange.c

```c
int minimumCoins(int amt)
{
   int coins = 0;

   coins += amt/100;
   amt %= 100;
   coins += amt/50;
   amt %= 50;
   coins += amt/20;
   amt %= 20;
   coins += amt/10;
   amt %= 10;
   coins += amt/5;
   amt %= 5;
   coins += amt/1; // retained for regularity
   amt %= 1;       // retained for regularity

   return coins;
}
```

Q: Can we do better?

2. Motivation #2: Vote Counting (1/2)

- A student election has just completed with 1000 votes casted for the three candidates: Tom, Dick and Harry.
- Write a program VoteCount.c to read in all the votes and display the total number of votes received by each candidate. Each vote has one of three possible values:
  - 1 for Tom
  - 2 for Dick
  - 3 for Harry

2. Motivation #2: Vote Counting (2/2)

Week7_VoteCount.c

```
#include <stdio.h>
#define NUM_VOTES 1000    // number of votes

int main(void)
{
    int i, vote, tom = 0, dick = 0, harry = 0;
    printf("Enter votes:\n");
    for (i = 0; i < NUM_VOTES; i++)
    {
        scanf("%d", &vote);
        switch (vote)
        {
            case 1: tom++; break;
            case 2: dick++; break;
            case 3: harry++; break;
        }
    }
    printf("Tom: %d; Dick: %d; Harry: %d\n", tom, dick, harry);
    return 0;
}
```

Q: What if there are 30 instead of 3 candidates?

2. Motivation #2: With 30 Candidates

```
#include <stdio.h>
#define NUM_VOTES 1000     // number of votes

int main(void)
{
    int i, vote, c1 = 0, c2 = 0, ..., c30 = 0;
    printf("Enter votes:\n");
    for (i = 0; i < NUM_VOTES; i++)
    {
        scanf("%d", &vote);
        switch (vote)
        {
            case 1: c1++; break;
            case 2: c2++; break;
            . . .
            case 30: c30++; break;
        }
    }
    . . .
}
```

!

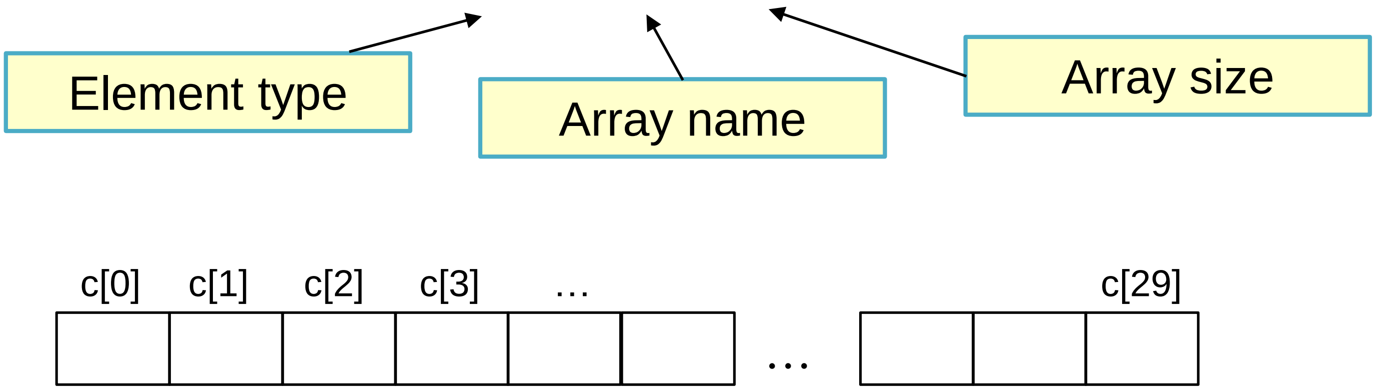
!

Q: Can we do better?

3. Introducing Array (1/2)

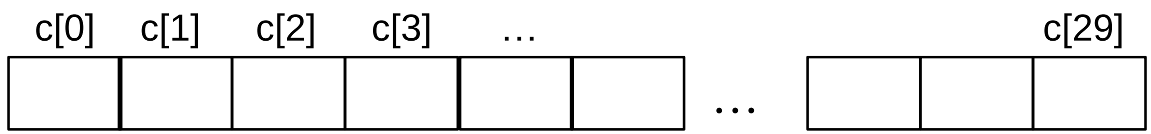
- It's inconvenient to define and use a set of variables c1, c2, ..., c30 in the previous example.
- Let's study a new language feature called ARRAY for batch processing of information.

```
int c[30];
```



3. Introducing Array (2/2)

```
int c[30];
```



- For the previous vote counting problem
  - c[0] will hold the number of votes for 1st candidate
  - c[1] holds the number of votes for 2nd candidate
  - …
  - c[29] for the 30th candidate.
- If we read in one more vote for candidate 4, we should increase c[3] by 1.

Q: Why increase c[3] by 1?

3.1 Array Declaration: Syntax

T arr [E]

- **arr** is the name of array
- E is an integer constant expression with a positive value
- T is a type (e.g., int, double, float, char…)
  - All array elements will be of the same type T
- Examples:

```
#define M 5
#define N 10

double foo[M*N+8]; // size of array foo is 58
char arr[10];      // this is good

int i;
float bar[i];      // DISCOURAGED!
```

Variable-length array is not supported by ISO C90 standard. gcc –pedantic gives warning.

3.2 Array Declarations w/ Initializers

- Array can be initialized at the same time of declaration.

```
int a[3] = {54, 9, 10}; // a[0]=54, a[1]=9, a[2]=10

int b[] = {1, 2, 3};
// size of b is 3 with b[0]=1, b[1]=2, b[2]=3

int c[5] = {17, 3, 10}; // partial initialization
// c[0]=17, c[1]=3, c[2]=10, c[3]=0, c[4]=0
```

- The following initializations are incorrect:

```
int e[2] = {1, 2, 3};  // warning issued: excess elements

int f[5];
f[5] = {8, 23, 12, -3, 6}; // too late to do this;
                           // compilation error
```

3.3 Demo #1: Using Array Variables

Fuller code two slides later.

```
#include <stdio.h>
#define NUM_VOTES 1000      // number of votes
#define NUM_CANDIDATES 30   // number of candidates
int main(void)
{
    int i, vote;
    int cand[NUM_CANDIDATES];

    for (i = 0; i < NUM_CANDIDATES; i++) // init array
        cand[i] = 0;

    printf("Enter votes:\n");
    for (i = 0; i < NUM_VOTES; i++)
    {
        scanf("%d", &vote);
        cand[vote-1]++;
    }
    . . .
}
```

Note

3.4 Vote Counting using Array

Week7_VoteCountArray.c

```
#define NUM_VOTES 1000      // number of votes
#define NUM_CANDIDATES 30   // number of candidates

int main(void)
{
    int i, vote, cand[NUM_CANDIDATES];
    for (i = 0; i < NUM_CANDIDATES; i++) { cand[i] = 0; }

    printf("Enter votes:\n");
    for (i = 0; i < NUM_VOTES; i++)
    {
        scanf("%d", &vote); // assume user enters valid data
        cand[vote-1]++; // add one more vote to candidate
    }
    for (i = 0; i < NUM_CANDIDATES; i++)
        printf("candidate %d: total %d, %.2f%%\n",
                i+1, cand[i], (cand[i] * 100.0)/NUM_VOTES);
    return 0;
}
```

Q: What is `%%`?

```
(data input skipped ...)
candidate 1: total 4, 4.01%
candidate 2: total 12, 12.03%
...
```

3.5 Demo #2: Using Array Initializer

- Modify the program to use array initializer.

Week7_VoteCountArrayVer2.c

```
#define NUM_VOTES 1000     // number of votes
#define NUM_CANDIDATES 30 // number of candidates
int main(void) {
    int i, vote, cand[NUM_CANDIDATES];
    for (i = 0; i < NUM_CANDIDATES; i++) { cand[i] = 0; }
    int cand[NUM_CANDIDATES] = { 0 };

    printf("Enter votes:\n");
    for (i = 0; i < NUM_VOTES; i++) {
        scanf("%d", &vote);
        cand[vote-1]++;
    }
    for (i = 0; i < NUM_CANDIDATES; i++)
        printf("candidate %d: total %d, %.2f%%\n",
                   i+1, cand[i], (cand[i] * 100.0)/NUM_VOTES);
    return 0;
}
```

3.6 Demo #3: Coin Change Revisit (1/2)

Algorithm 1:

```
input: amt (in cents); output: coins
coins ← 0
coins += amt/100; amt %=100;
coins += amt/50; amt %= 50;
coins += amt/20; amt %= 20;
coins += amt/10; amt %= 10;
coins += amt/5; amt %= 5;
coins += amt/1; amt %= 1;
print coins
```

Algorithm 2:

```
input: amt (in cents); output: coins
coins ← 0
From the largest denomination to the smallest:
    coins += amt/denomination
    amt %= denomination
    go to next denomination
print coins
```

Q: how can we easily switch from one denomination to another?

array!

Algorithm 3:

input: amt (in cents); output: coins

coins ← 0

for i from 0 to 5 // there are 6 denominations

    coins += amt/D_i // D_0, D_1, D_2, D_3, D_4, D_5

    amt %= D_i

print coins

3.6 Demo #3: Coin Change Revisit (2/2)

```c
int minimumCoins(int amt)
{
   int coins = 0;
   coins += amt/100;
   amt %= 100;
   coins += amt/50;
   amt %= 50;
   coins += amt/20;
   amt %= 20;
   coins += amt/10;
   amt %= 10;
   coins += amt/5;
   amt %= 5;
   coins += amt/1;
   amt %= 1;
   return coins;
}
```

Week7_CoinChange.c

```c
int minimumCoins(int amt)
{
   int denoms[] = {100,50,20,10,5,1};
   int i, coins = 0;

   for (i=0; i<6; i++)
   {
      coins += amt/denoms[i];
      amt %= denoms[i];
   }

   return coins;
}
```

Week7_CoinChangeArray.c

Q: which version is better?

4. Array Assignment (1/2)

- The following is illegal in C:

```
#define N 10
int source[N] = { 10, 20, 30, 40, 50 };
int dest[N];
dest = source;  // illegal!
```

Q: Why?

source[0] … source[9]

10	20	30	40	50	0	0	0	0	0

dest[0] … dest[9]

?	?	?	?	?	?	?	?	?	?

A: array name refers to the address of the first element.

4. Array Assignment (2/2)

- Method 1: Use a loop

```
int i;
for (i = 0; i < 10; i++)
   dest[i] = source[i];
```

source[0] ... source[9]

10	20	30	40	50	0	0	0	0	0

dest[0] ... dest[9]

10	20	30	40	50	0	0	0	0	0

- Method 2: Use C library function **memcpy()**
  - `#include <string.h>`
  - Out of the scope of CS1010

5. Use Array in Function Calls

Week7_SumArray.c

```
#include <stdio.h>
int sumArray(int [], int); // function prototype

int main(void) {
    int foo[8] = {5, 3, 7, 1, -4, 2};
    int bar[] = {2, 4, 6};
    printf("sum is %d\n", sumArray(foo, 8));
    printf("sum is %d\n", sumArray(foo, 3));
    printf("sum is %d\n", sumArray(bar, 3));
    return 0;
}

// need an array size parameter separately
int sumArray(int arr[], int size) {
    int i, total=0;
    for (i=0; i<size; i++)
        total += arr[i];
    return total;
}
```

Q: What is the output?

```
sum is 14
sum is 15
sum is 12
```

Q: How about this function call?

```
sumArray(bar, 5)
```

6. Passing Array Arguments (1/4)

- Caution!
  - When passing a value representing the number of array elements to be processed, that value must not exceed the actual array size.

```
printf("sum is %d\n", sumArray(foo, 10));
```

Compiler won't detect this "error".

  - There is NO boundary checking done by the compiler.

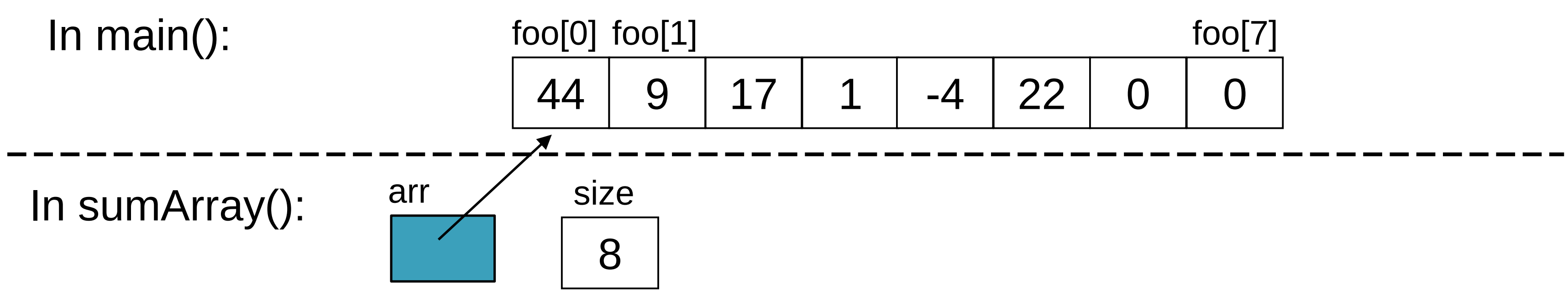
6. Passing Array Arguments (2/4)

```
int main(void) {
    ...
    printf("sum is %d\n", sumArray(foo, 8));
    ...
}
int sumArray(int arr[], int size) {
    ...
}
```

```
sumArray(foo[], 8)
```

Wrong!

- Recall that array name is the address of its first element.



6. Passing Array Arguments (3/4)

- Alternative syntax
  - The following shows the alternative syntax for array parameter in function prototype and function header

```
int sumArray(int *, int); // function prototype
```

```
int sumArray(int *arr, int size) { ... }
```

- However, we recommend the [ ] notation

```
int sumArray(int [], int); // function prototype
```

```
int sumArray(int arr[], int size) { ... }
```

6. Passing Array Arguments (4/4)

- Function prototype
  - As mentioned, name of parameters are optional. Hence, both of the followings are acceptable and equivalent:

```
int sumArray(int [], int);
```

```
int sumArray(int arr[], int size);
```

- Function header
  - No need to put array size inside [ ]; even if array size is present, compiler just ignores it.
  - Instead, provide the array size through another parameter.

```
int sumArray(int arr[], int size) { ... }
```

```
int sumArray(int arr[8], int size) { ... }
```

Ignored by compiler

Actual number of elements you want to process

7. Modifying Array Arguments (1/2)

Week7_ModifyArrayArg.c

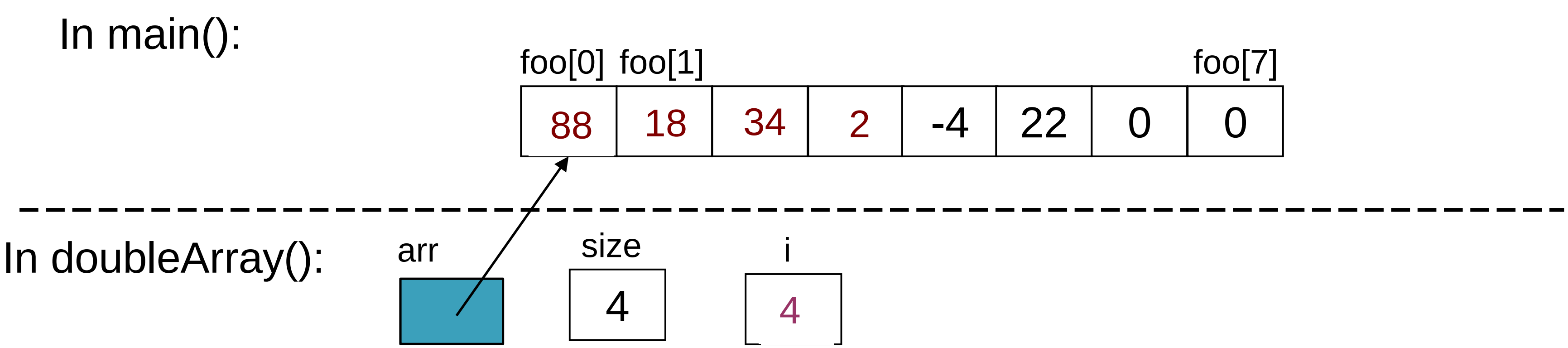
```c
// preprocessor directives and
// function prototypes omitted
int main(void) {
    int foo[8] = {44, 9, 17, 1, -4, 22};
    doubleArray(foo, 4);
    printArray(foo, 8);
    return 0;
}
// double the values of array elements
void doubleArray(int arr[], int size) {
    int i;
    for (i=0; i<size; i++)
        arr[i] *= 2;
}
// print arr
void printArray(int arr[], int size) {
    int i;
    for (i=0; i<size; i++)
        printf("%d ", arr[i]);
    printf("\n");
}
```

Q: What is the output?

```
88 18 34 2 -4 22 0 0
```

7. Modifying Array Arguments (2/2)

```c
int main(void) {
    int foo[8] = {44, 9, 17, 1, -4, 22};
    doubleArray(foo, 4);
    . . .
}
// double the values of array elements
void doubleArray(int arr[], int size) {
    int i;
    for (i=0; i<size; i++)
        arr[i] *= 2;
}
```



8. Exercise #2: Set Containment

- Consider two arrays arrA and arrB of <u>distinct</u> *int* values, where their sizes are sizeA and sizeB respectively (less than 10).
- Write a function

  int isSubset(int arrA[], int sizeA, int arrB[], int sizeB)

  to check if numbers in arrA is a subset of numbers in arrB.

  This function returns 1 if so, 0 otherwise.

Algorithm!

- Skeleton:

```
cp ~cs1010/lecture/Week7_SetContainment.c .
```

- Sample run:

```
Size of 1st array? 4
        Enter 4 values: 14 5 1 9
Size of 2nd array? 7
        Enter 7 values: 2 9 3 14 5 6 1
1st array is a subset of 2nd array
```

Searching and Sorting

- We will study some simple yet useful classical algorithms which find their place in many CS applications.
  - Searching for some data amid very large collection of data
  - Sorting very large collection of data according to some order

- We will begin with an algorithm (idea), then show how the algorithm is transformed into a C program (implementation).
  - This brings back (reminds you):

    the importance of beginning with an algorithm

1. Searching (1/2)

- Searching is a common task that we carry out without much thought everyday.
  - Searching for a location in a map.
  - Searching for the contact of a particular person.
  - Searching for a nice picture for your project report.
  - Searching for a research paper required in your course.
  - etc.

- In this lecture, you will learn how to search for an item (sometimes called a search key) in an array.

1. Searching (2/2)

- Problem statement:

  Given a list (collection of data) and a search key *X*, return the position of *X* in the list if it exists.

  For simplicity, we shall assume there are no duplicate values in the list.

- We will count the number of comparisons the algorithms make to analyze their performance.
  - The ideal searching algorithm will make the least possible number of comparisons to locate the desired data.
  - We will introduce worst-case scenario.
    - (This topic is called analysis of algorithms, which will be formally introduced in CS1020. Here, we will give an informal introduction just for an appreciation.)

2. Linear Search (1/3): Algorithm

- Also known as Sequential Search
- Idea: Search the list from one end to the other end in linear progression.
- Algorithm:

```
// Search for key in list A with n items
linear_search (A, n, key)
{
    for i = 0 to n-1
        if A_i is key then report i
}
```

Example: Search for 24 in this list

87	12	51	9	24	63
no	*no*	*no*	*no*	*yes!*	

Q: What to report if key is not found? (aim for a clean design)

2. Linear Search (2/3): Code

- If the list is an array, how would you implement the Linear Search algorithm?

```
// To search for key in arr using linear search
// Return index if found; otherwise return -1
int linearSearch(int arr[], int size, int key)
{
    int i;
    for (i=0; i<size; i++)
        if (key == arr[i])
            return i;
    return -1;  // not found
}
```

Q: What would be returned if array contains duplicated values of the key?

2. Linear Search (3/3): Performance

- We use the number of comparisons here as a rough measurement.
  - Analysis can be done for best case, average case, and worst case. We will focus on the worst case.
- Given an array with *n* elements, in the worst case,

```
int linearSearch(int arr[], int n, int key)
{
    int i;
    for (i=0; i<n; i++)
        if (key == arr[i])
            return i;
    return -1;
}
```

Q: What is the maximum number of comparisons in this algorithm?

n comparisons

Q: Under what circumstances do we encounter the worst case?

(a) Key not found
(b) Found at last position

3. Binary Search (1/6)

- The idea is simple and fantastic, but applied on the searching problem, it has this pre-condition that the list must be sorted before-hand.
- How the data is organized (in this case, sorted) usually affects how we choose/design an algorithm to access them.
- In other words, sometimes (actually, very often) we seek out new way to organize the data so that we can process them more efficiency.

3. Binary Search (2/6): Algorithm

(Pre-condition: list is sorted in ascending order)

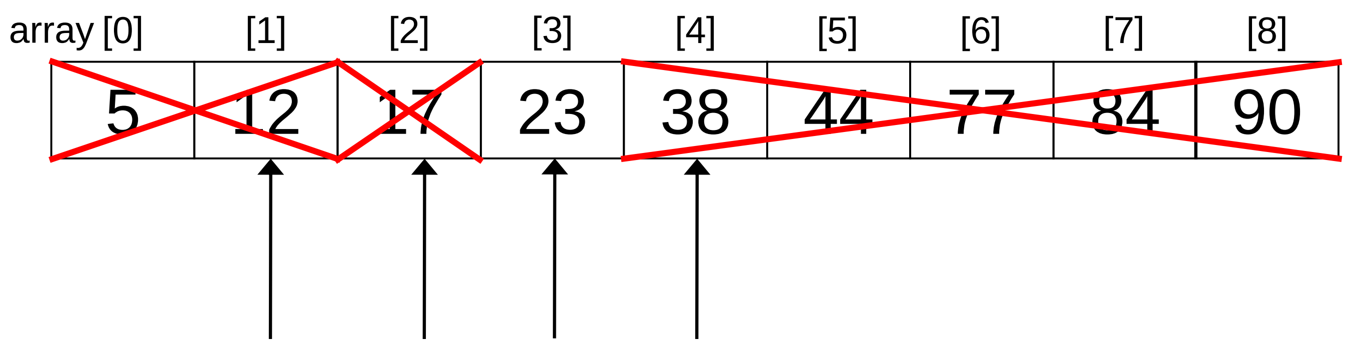
- Algorithm
  - Look for the *key* in the middle position of the list.

    Either of the following 2 cases happens:

    - If the *key* is **smaller** than the middle element, then "discard" the right half of the list and repeat the process.
    - If the *key* is **greater** than the middle element, then "discard" the left half of the list and repeat the process.
  - Terminating condition: either the *key* is found, or when all elements have been "discarded".

3. Binary Search (3/6): Illustration

- Example: Search for *key* = 23



1. low = 0, high = 8, mid = (0+8)/2 = 4
2. low = 0, high = 3, mid = (0+3)/2 = 1
3. low = 2, high = 3, mid = (2+3)/2 = 2
4. low = 3, high = 3, mid = (3+3)/2 = 3

Found!
Return 3

3. Binary Search (4/6): Iterative Code

- Iterative version

```
// To search for key in sorted arr using binary search
// Return index if found; otherwise return -1
int binarySearch(int arr[], int size, int key)
{
    int low=0, high=size-1, mid=(low + high)/2;
    while ((low <= high) && (arr[mid] != key))
    {
        if (key < arr[mid])
            high = mid - 1;
        else
            low = mid + 1;
        mid = (low + high)/2;
    }
    if (low > high) return -1;
    else return mid;
}
```

Week10_BinarySearch.c

3. Binary Search (5/6): Analysis

- In binary search, each step eliminates the problem size (array size) by half.
  - The problem size gets reduced to 1 very quickly (see next slide)
- This is a simple yet powerful strategy, of halving the solution space in each step
  - Which exercise employs a similar strategy?

A: Bisection Method

- Such strategy, a special case of divide-and-conquer paradigm, can be naturally implemented using recursion.
- At the moment, we will stick to repetition (loop)
  - You can write a recursion version after saturday's lecture.

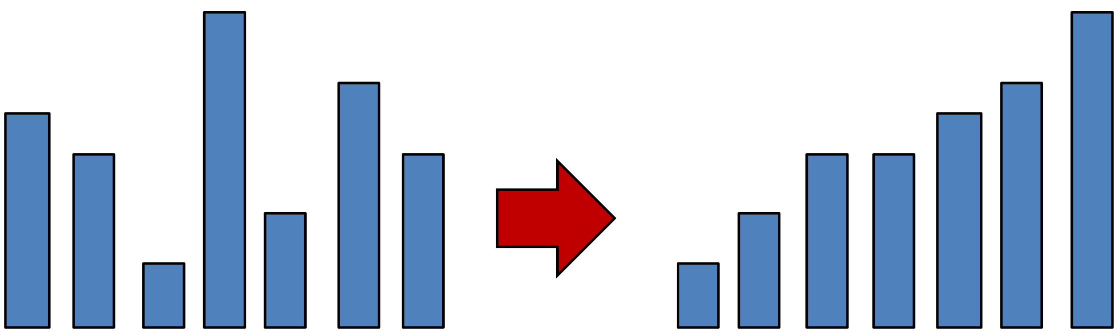
3. Binary Search (6/6): Performance

- In binary search, each step eliminates the problem size (array size) by half.
  - The problem size gets reduced to 1 very quickly.
- Worst-case analysis

Array size n	Linear Search (n comparisons)	Binary search ($\log_2 n$ comparisons)
100	100	≈7
1,000	1,000	≈10
10,000	10,000	≈14
100,000	100,000	≈17
1,000,000	1,000,000	≈20
10^9	10^9	≈30

4. Sorting (1/2)

- Sorting is any process of arranging items in some sequence.
- Sorting is important because once a set of items is sorted, many problems (such as searching) become easy.
  - Searching can be speeded up.
  - Determining whether the items in a set are all unique.
  - Finding the median item in the set.
  - etc.



4. Sorting (2/2)

- Problem statement:

  Given a list of *n* items, arrange all items into ascending order.

- We will implement the list as an integer array.
- We will introduce <u>two</u> basic sort algorithms.
- We will count the number of comparisons the algorithms make to analyze their performance.
  - The ideal sorting algorithm will make the least possible number of comparisons to arrange data in a designated order.
- We will compare the algorithms by analyzing their worst-case performance.

5. Selection Sort (1/3)

- Algorithm
  - Step 1: Find the smallest element in the list.
  - Step 2: Swap this smallest element with the element in the first position. (Now, the smallest element is in the right place.)
  - Step 3: Repeat steps 1 and 2 with the list having one fewer element (i.e. the smallest element just found and placed is "exempted" from further processing).

5. Selection Sort (2/3)

$n = 9$

	array[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
	first		min						
1st pass:	23	17	5	90	12	44	38	84	77
		first			min				
2nd pass:	5	17	23	90	12	44	38	84	77
			first		min				
3rd pass:	5	12	23	90	17	44	38	84	77
				first	min				
4th pass:	5	12	17	90	23	44	38	84	77

5. Selection Sort (3/3)

Q: How many passes for an array with n elements?

$n-1$

$n = 9$

first min

5th pass:	5	12	17	23	90	44	38	84	77

first min

6th pass:	5	12	17	23	38	44	90	84	77

first min

7th pass:	5	12	17	23	38	44	90	84	77

first min

8th pass:	5	12	17	23	38	44	77	84	90

Final array:	5	12	17	23	38	44	77	84	90

5. Demo #2: Selection Sort

Week10_SelectionSort.c

```
// To sort arr in increasing order
void selectionSort(int arr[], int size)
{
    int i, start_index, min_index, temp;

    for (start_index = 0; start_index < size-1; start_index++)
    {
        // each iteration of the for loop is one pass

        // find the index of minimum element
        min_index = start_index;
        for (i = start_index+1; i < size; i++)
            if (arr[i] < arr[min_index])
                min_index = i;

        // swap minimum element with element at start_index
        temp = arr[start_index];
        arr[start_index] = arr[min_index];
        arr[min_index] = temp;
    }
}
```

5. Selection Sort: Performance

- We choose the number of comparisons as our basis of analysis.
- Comparisons of array elements occur in the inner loop, where the minimum element is determined.
- Assuming an array with n elements. Table below shows the number of comparisons for each pass.
- The total number of comparisons is calculated in the formula below.
- Such an algorithm is call an n^2 algorithm, or quadratic algorithm, in terms of running time complexity.

Pass	#comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
…	…
$n - 1$	1

$$\boxed{\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} \cong n^2}$$

6. Bubble Sort

- Selection sort makes one exchange at the end of each pass.
- What if we make more than one exchange during each pass?
- The key idea of the bubble sort is to make pairwise comparisons and exchange the positions of the pair if they are out of order.

6. One Pass of Bubble Sort

0	1	2	3	4	5	6	7	8
23	17	5	90	12	44	38	84	77

exchange

17	23	5	90	12	44	38	84	77

exchange

17	5	23	90	12	44	38	84	77

ok exchange

17	5	23	12	90	44	38	84	77

exchange

17	5	23	12	44	90	38	84	77

exchange

17	5	23	12	44	38	90	84	77

exchange

17	5	23	12	44	38	84	90	77

exchange

17	5	23	12	44	38	84	77	90

Done!

Q: Is the array sorted?
Q: What did we achieve?

6. Demo #3: Bubble Sort

Week10_BubbleSort.c

```
// To sort arr in increasing order
void bubbleSort(int arr[], int size)
{
    int i, limit, temp;

    for (limit = size-2; limit >= 0; limit--)
    {
        // limit is where the inner loop variable i should end

        for (i=0; i<=limit; i++) // one pass
        {
            if (arr[i] > arr[i+1])  // swap arr[i] with arr[i+1]
            {
                temp = arr[i];
                arr[i] = arr[i+1];
                arr[i+1] = temp;
            }
        }
    }
}
```

6. Bubble Sort: Performance

- Bubble sort, like selection sort, requires $n - 1$ passes for an array with n elements.
- The comparisons occur in the inner loop. The number of comparisons in each pass is given in the table below.
- The total number of comparisons is calculated in the formula below.
- Like selection sort, bubble sort is also an n^2 algorithm, or quadratic algorithm, in terms of running time complexity.

Pass	#comparisons
1	$n - 1$
2	$n - 2$
3	$n - 3$
…	…
$n - 1$	1

$$\sum_{i=1}^{n-1} i = \frac{(n-1)(n)}{2} = \frac{n^2 - n}{2} \cong n^2$$

6. Bubble Sort: Enhanced Version

- It is possible to enhance bubble sort algorithm to reduce the number of passes.
  - Suppose that in a certain pass, no swap is needed. This implies that the array is already sorted, and hence the algorithm may terminate without going on to the next pass.

7. More Sorting Algorithms

- What we have introduced are 2 basic sort algorithms. Together with the Insertion Sort algorithm, these 3 algorithms are the simplest.
- However, they are very slow, as their running time complexity is quadratic.
- Faster sorting algorithms exist and are topics in more advanced programming modules.
  - Merge sort (CS1020)
  - Quick sort (CS1020)
  - Heap sort (CS2010)

8. Animated Sorting Algorithms

- There are a number of animated sorting algorithms on the Internet.

- Here are two sites:
  - http://www.sorting-algorithms.com/
  - http://www.cs.ubc.ca/~harrison/Java/sorting-demo.html

- YouTube video on Bubble sort:
  - http://www.youtube.com/watch?v=lyZQPjUT5B4

The End