Introduction to Computer Science
Lecture 12

Nazareno Aguirre

(based on material by Guillaume Hoffmann)

Today's topic

Time complexity

Time complexity

- Consider an algorithm that takes as input an array of size n
- We express **running time** of the algorithm as a function of n
  - short sequences are easier to sort than long ones
- Generally, we seek **upper bounds** on the running time
- Worst-case:
  - T(n) = maximum time of the algorithm on **any** input of size n
- Average-case (not to be covered in this course):
  - T(n) = **expected** time of algorithm over **all** inputs of size n

Machine-independent time

What is `selectionSort()`'s worst-case time, in seconds?

- It depends on the characteristics of our computer

Idea:

- Ignore machine-dependent constants.
- Do not talk about "how many seconds", talk about "how many steps" or "how many operations".
- Look at *growth* of T(n) as $n \to \infty$.

This is **asymptotic analysis**.

Worst Case Asymptotic Analysis

For the algorithm to be analyzed,

- Determine what is the “size” of the input
- Determine which are the relevant operations/steps
- Analyze the worst case configuration among all possible concrete inputs of arbitrary size n (worst case is that which would exercise more times the relevant operations)
- Express the number of relevant operations in the worst case, as a function T on the input size n
- Determine the asymptotic growth of T

Worst Case Asymptotic Analysis

```
void insertionSort(int array[], int size) {
    for (int i = 1; i < size; i++) {
        int j = i;
        while (j > 0 && array[j-1] > array[j]) {
            int aux = array[j];
            array[j] = array[j-1];
            array[j-1] = aux;
        }
    }
}
```

Worst Case Asymptotic Analysis

```
void selectionSort(int array[], int size) {
    for (int i = 0; i < size-1; i++) {
        int min_index = i;
        for (int j = i+1; j < size; j++) {
            if (array[j] < array[min_index]) {
                min_index = j;
            }
        }
        int aux = array[i];
        array[i] = array[min_index];
        array[min_index] = aux;
    }
}
```

Θ-notation (big theta)

$\Theta(g(n))$ = { f(n) : there exist positive constants c_1, c_2, n_0 such that

$$0 \leq c_1.g(n) \leq f(n) \leq c_2.g(n) \text{ for all } n \geq n_0 \}$$

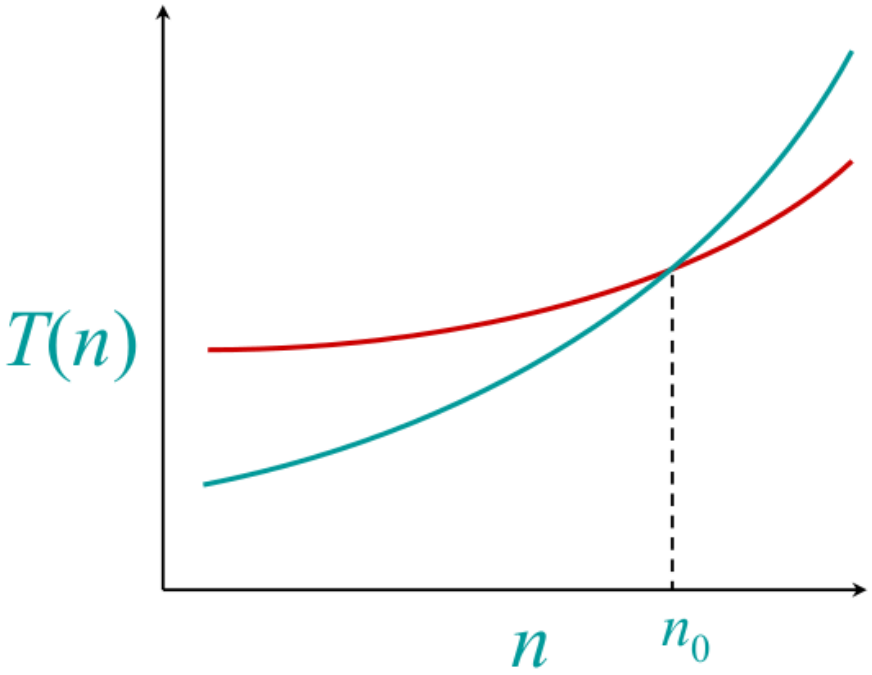
"f is bounded both above and below by g asymptotically "

In practice:

- Drop low-order terms; ignore leading constants.
- Example: $3n^3 + 90n^2 – 5n + 6046 = \Theta(n^3)$

Asymptotic performance

When n gets large enough, a $\Theta(n^2)$ algorithm is always better than a $\Theta(n^3)$ algorithm.

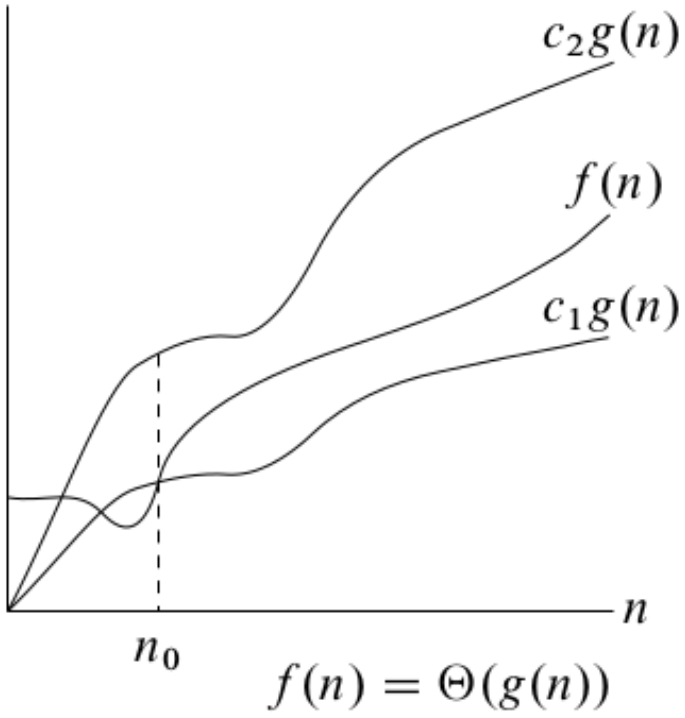


- Asymptotic analysis is a useful tool
- It helps us compare the efficiency of alternative algorithms for a given problem.
- It can also allow us to estimate the worst case running time of algorithms for larger inputs, from the actual running times of smaller inputs

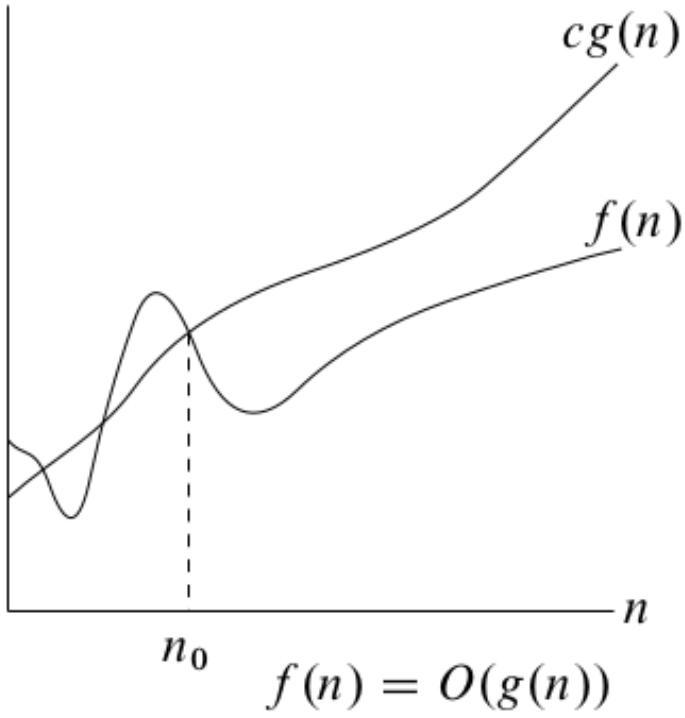
Difference between big-theta Θ and big-oh O

- We are usually interested in searching for tight bounds.
- Θ-notation refers to a *tight bound*:
  - $3n^3 + 90n^2 – 5n + 6046$ belongs to $\Theta(n^3)$
- There is another notation, O-notation, which refers to an *upper bound*:
  - $3n^3 + 90n^2 – 5n + 6046$ belongs to $O(n^{10})$ // true but not very informative
  - $3n^3 + 90n^2 – 5n + 6046$ belongs to $O(n^5)$ // better but not ideal
  - $3n^3 + 90n^2 – 5n + 6046$ belongs to $O(n^3)$ // the tightest possible

Difference between big-theta Θ and big-oh O



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



$f(n) = O(g(n))$

Big-theta Θ vs. big-oh O, worst-case vs. all cases

- We often use O-notation to describe the running time of an algorithm by a more immediate upper bound
- For example, the doubly nested loop structure of `selectionSort()` immediately yields an $O(n^2)$ upper bound on the worst-case running time.
- The O-notation describes an upper bound, so when we use it to bound the worst-case running time of an algorithm, we have a bound on the running time of the algorithm on *every* input.
- Thus, the $O(n^2)$ bound on worst-case running time of `selectionSort()` also applies to its running time on every input.
- However, a $\Theta(n^2)$ bound on the worst-case running time, does not imply a $\Theta(n^2)$ bound on the running time on every input.
- For instance, `insertionSort()` has an $\Theta(n^2)$ bound for the worst-case, but if the array is already sorted, it runs in linear time

Complexity naming

- O(1): constant time
- O(log n): logarithmic time
- O(n): linear time
- $O(n^2)$: quadratic time
- $O(n^3)$: cubic time
- $O(2^n)$: exponential time

Accidentally quadratic code

```
void toLower(char s[]) {
    for (int i = 0; i < strlen(s); i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] += 'a' - 'A';
        }
    }
}
```

```
int strlen(const char s[]) {
    int len = 0;
    while (s[len] != '\0') {
        len++;
    }
    return len;
}
```

- Loop condition does a call to `strlen()`, which is O(n) time
- In O(n) time:

```
void toLower(char s[]) {
    for (int i = 0; s[i] != '\0'; i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] += 'a' - 'A';
        }
    }
}
```

```
void toLower(char s[]) {
    int n = strlen(s);
    for (int i = 0; i < n; i++) {
        if (s[i] >= 'A' && s[i] <= 'Z') {
            s[i] += 'a' - 'A';
        }
    }
}
```

Algorithms review with time complexity

- Summing elements of a sequence: O(n)
- Fisher-Yates Shuffle: O(n)
- Insertion Sort: $O(n^2)$
- Selection Sort: $O(n^2)$
- Merge: O(n + m) (n and m are the lengths of the sequences to merge)
- Linear search in a sequence: O(n)
- Binary search in sorted **array**: O(log n)

Quadratic hasDuplicate()

Design a program `hasDuplicate(int a[], int n)`, that runs in $O(n^2)$ time and returns true if the array contains a duplicated value (two or more times), false otherwise.

For instance, {1,2,3,4,5,6} has no duplicate, {1,2,3,4,5,2} has duplicate.

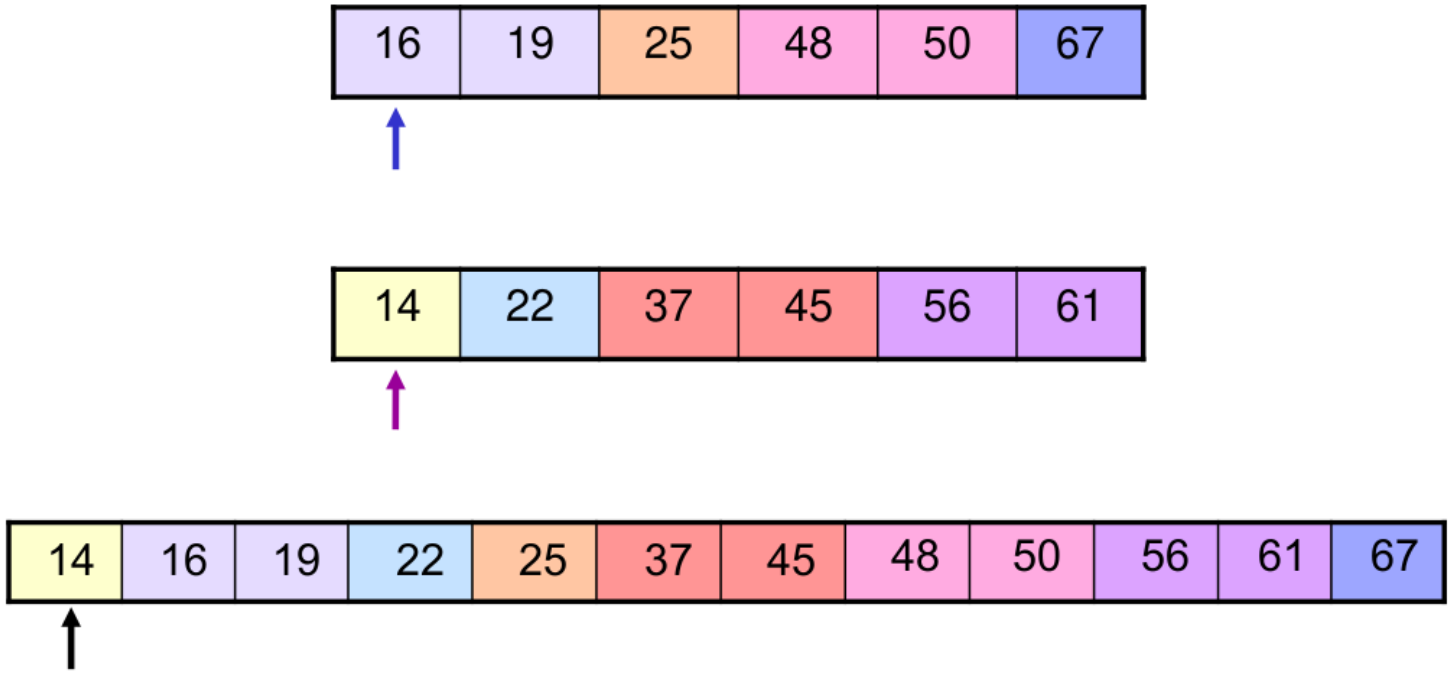
Cubic hasTriplicate()

Design a program `hasTriplicate(int a[], int n)`, that runs in $O(n^3)$ time and returns true if the array contains a value that appears thrice (or more times), 0 otherwise.

For instance:

- {1,2,3,4,5,6} has no triplicate
- {1,2,3,4,5,2} has no triplicate
- {1,**2**,3,**2**,5,**2**} has a triplicate

Merge()



Using merge() to create a Sorting Algorithm

Idea of merge sort:

have array a[] as input

merge pairs of elements in array:

a[0] a[1] => get a[0..1] sorted
a[2] a[3] => get a[2..3] sorted
...

then merge groups of 4:

a[0..1] a[2..3]=> get a[0..3] sorted
...
then groups of 8:...
....
then groups of n/2 => a[] sorted

mergeSort()

```
void mergeSort(int array[], int n) {
  for (int curr_size = 1; curr_size < n; curr_size = 2 * curr_size) {
       for (int left_start = 0; left_start < n-1; left_start += 2*curr_size) {
           int mid = min(left_start + curr_size - 1, n-1);
            int right_end = min(left_start + 2*curr_size - 1, n-1);
           merge(arr, left_start, mid, right_end);
       }
   }
}
```

Time Complexity of mergeSort()

- For each value of curr_size (outer for loop):
  - sum of all `merge()` calls on subarrays whose total size is n: O(n)
- Outer loop repeats (log n) times
- Total time: O(n log n)

```
void mergeSort(int array[], int n) {
   int left_start;
   for (int curr_size = 1; curr_size < n; curr_size = 2 * curr_size) {
       for (int left_start = 0; left_start < n-1; left_start += 2*curr_size) {
           int mid = min(left_start + curr_size - 1, n-1);
            int right_end = min(left_start + 2*curr_size - 1, n-1);
           merge(arr, left_start, mid, right_end);
       }
   }
}
```

Recursive mergeSort()

```
void mergeSort(int arr[], int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;

        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);

        merge(arr, left, mid, right);
    }
}
```

Recursive mergeSort() (cont.)

```
void merge(int arr[], int left, int mid, int right) {
    int i, j, k;
    int n1 = mid - left + 1;
    int n2 = right - mid;
    int leftArr[n1], rightArr[n2];
    for (i = 0; i < n1; i++)
        leftArr[i] = arr[left + i];
    for (j = 0; j < n2; j++)
        rightArr[j] = arr[mid + 1 + j];
    i = 0;
    j = 0;
    k = left;
    while (i < n1 && j < n2) {
        if (leftArr[i] <= rightArr[j]) {
            arr[k] = leftArr[i];
            i++;
        }
        else {
            arr[k] = rightArr[j];
            j++;
        }
        k++;
    }
    while (i < n1) {
        arr[k] = leftArr[i];
        i++;
        k++;
    }
    while (j < n2) {
        arr[k] = rightArr[j];
        j++;
        k++;
    }
}
```

Final Remarks

- O(n log n) is the optimal time complexity for sorting sequences of size n
  - merge sort is not the only sorting algorithm with that running time
- Asymptotic analysis is a first approach to solving problems efficiently
  - Later on you will see many low-level factors that also impact efficiency
- Time complexity is an important topic to be aware of when programming
- Careful with accidentally creating $O(n^2)$ code that could be O(n), or k*T(n) code that could be T(n).