Algorithm Analysis

data structure – a systematic way of organizing and accessing data

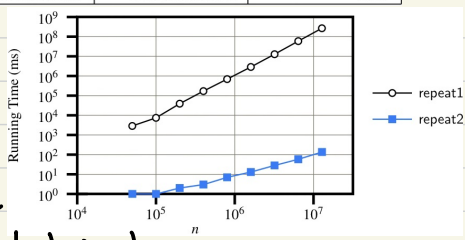
algorithm – a step-by-step procedure for performing some task in a finite amount of time

1. Experimental Studies

```java
/** Uses repeated concatenation to compose a String with n copies of character c. */
public static String repeat1(char c, int n) {
  String answer = "";
  for (int j=0; j < n; j++)
    answer += c;
  return answer;
}

/** Uses StringBuilder to compose a String with n copies of character c. */
public static String repeat2(char c, int n) {
  StringBuilder sb = new StringBuilder();
  for (int j=0; j < n; j++)
    sb.append(c);
  return sb.toString();
}
```

n	repeat1 (in ms)	repeat2 (in ms)
50,000	2,884	1
100,000	7,437	1
200,000	39,158	2
400,000	170,173	3
800,000	690,836	7
1,600,000	2,874,968	13
3,200,000	12,809,631	28
6,400,000	59,594,275	58
12,800,000	265,696,421	135



1.1. Chalenges

- experiment performed in same hardware and software
- experiment on limited set of test inputs
- fully implemented algorithm

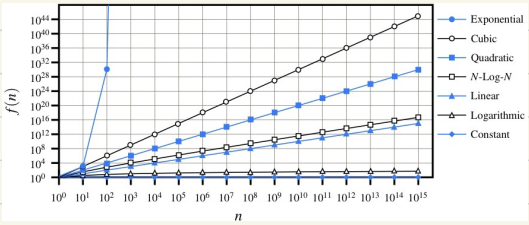
1.2. Beyond Experimental Analysis

- independent of hardware and software environment
- high-level description without implementation
- takes into account all possible inputs

1.2.1. Counting Primitive Operations

- assigning a value to a variable
- following an object reference
- performing an arithmetic operation
- comparing two numbers
- accessing a single element of an array by index
- calling a method
- returning from a method

2. The Seven Functions



2.1. The Constant Function

$f(n) = c \quad g(n) = 1 \quad f(n) = cg(n)$

2.2. The Logarithm Function

$f(n) = \log_b n, \quad b > 1$

2.3. The Linear Function

$f(n) = n$

2.4. The N-Log-N Function

$f(n) = n \log n$

2.5. The Quadratic Function

$f(n) = n^2$

2.6. The Cubic Function and Other Polynomials

$f(n) = n^3$

2.7. The Exponential Function

$f(n) = b^n$

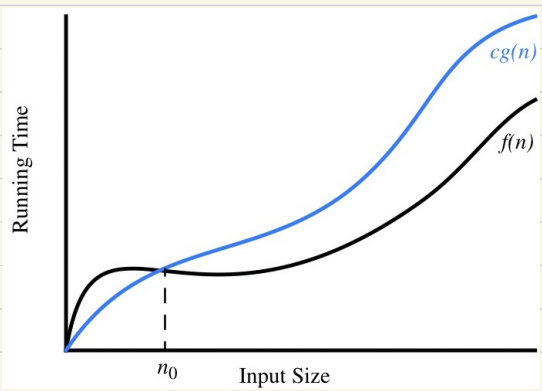
2.8. Comaparing Growth Rates

constant	logarithm	linear	n-log-n	quadratic	cubic	exponential
1	$\log n$	n	$n \log n$	n^2	n^3	a^n

3 Asymptotic Analysis

3.1. The "Big-Oh" Notation

$f(n)$ is $O(g(n))$ iff
$\exists c > 0, \exists n_0 \geq 1, n_0 \in \mathbb{Z}$ s.t.
$f(n) \leq cg(n)$ for $n \geq n_0$
$f(n)$ - polynomial, degree d
$\Rightarrow f(n)$ is $O(n^d)$



3.1.1. Big-Omega

$f(n)$ is $\Omega(g(n))$ iff
$\exists c > 0, \exists n_0 \geq 1, n_0 \in \mathbb{Z}$ s.t.
$f(n) \geq cg(n)$ for $n \geq n_0$

3.1.2. Big-Theta

$f(n)$ is $\Theta(g(n))$ iff $\exists c', c'' > 0$, $\exists n_0 \geq 1$, $n_0 \in \mathbb{Z}$ s.t.
$c' g(n) \leq f(n) \leq c'' g(n)$ for $n \geq n_0$

3.2. Comparative Analysis

```
1  /** Returns the maximum value of a nonempty array of numbers. */
2  public static double arrayMax(double[ ] data) {
3    int n = data.length;
4    double currentMax = data[0];                // assume first entry is biggest (for now)
5    for (int j=1; j < n; j++)                   // consider all other entries
6      if (data[j] > currentMax)                 // if data[j] is biggest thus far...
7        currentMax = data[j];                   // record it as the current max
8    return currentMax;
9  }
```

c'' | c'

$f(n) = c'(n-1) + c'' = c'n + (c'' - c')$
$\Rightarrow f(n) \leq c'n$
running time is $O(n)$

```
1  /** Uses repeated concatenation to compose a String with n copies of character c. */
2  public static String repeat1(char c, int n) {
3    String answer = "";
4    for (int j=0; j < n; j++)
5      answer += c;
6    return answer;
7  }
```

creates new string, answer pointing to it

$f(n) = \sum_{i=1}^{n} i = 1 + 2 + \ldots + n$
$\Rightarrow f(n)$ is $O(n^2)$

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint1(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        for (int c : groupC)
6          if ((a == b) && (b == c))
7            return false;                      // we found a common value
8    return true;                               // if we reach this, sets are disjoint
9  }
```

$O(n^3)$

```
1  /** Returns true if there is no element common to all three arrays. */
2  public static boolean disjoint2(int[ ] groupA, int[ ] groupB, int[ ] groupC) {
3    for (int a : groupA)
4      for (int b : groupB)
5        if (a == b)                            // only check C when we find match from A and B
6          for (int c : groupC)
7            if (a == c)                        // and thus b == c as well
8              return false;                    // we found a common value
9    return true;                               // if we reach this, sets are disjoint
10 }
```

$O(n^2)$

```
1  /** Returns true if there are no duplicate elements in the array. */
2  public static boolean unique1(int[ ] data) {
3    int n = data.length;
4    for (int j=0; j < n-1; j++)
5      for (int k=j+1; k < n; k++)
6        if (data[j] == data[k])
7          return false;                        // found duplicate pair
8    return true;                               // if we reach this, elements are unique
9  }
```

$O(n^2)$

```
1  /** Returns true if there are no duplicate elements in the array. */
2  public static boolean unique2(int[ ] data) {
3    int n = data.length;
4    int[ ] temp = Arrays.copyOf(data, n);      // make copy of data
5    Arrays.sort(temp);                         // and sort the copy
6    for (int j=0; j < n-1; j++)
7      if (temp[j] == temp[j+1])                // check neighboring entries
8        return false;                          // found duplicate pair
9    return true;                               // if we reach this, elements are unique
10 }
```

$O(n \log n)$

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[ ] prefixAverage1(double[ ] x) {
3    int n = x.length;                          O(1)
4    double[ ] a = new double[n];               O(n)
5    for (int j=0; j < n; j++) {                O(n^2)
6      double total = 0;                        O(n)
7      for (int i=0; i <= j; i++)               O(n^2)
8        total += x[i];                         O(n^2)
9      a[j] = total / (j+1);                    O(n)
10   }
11   return a;                                  O(1)
12 }
```

```
1  /** Returns an array a such that, for all j, a[j] equals the average of x[0], ..., x[j]. */
2  public static double[ ] prefixAverage2(double[ ] x) {
3    int n = x.length;                          O(1)
4    double[ ] a = new double[n];               O(n)
5    double total = 0;                          O(1)
6    for (int j=0; j < n; j++) {                O(n)
7      total += x[j];
8      a[j] = total / (j+1);
9    }
10   return a;                                  O(1)
11 }
```

$O(n)$

4. Simple Justification Techniques

4.1. Counterexample

4.2. Contrapositive & Contradiction

4.3 Induction & Loop Invariants