Algorithm Efficiency

There are often many approaches (algorithms) to solve a problem. How do we choose between them?

At the heart of computer program design are two (sometimes conflicting) goals:

1. To design an algorithm that is easy to understand, code and debug.
2. To design an algorithm that makes efficient use of the computer's resources.

Goal (1) is the concern of Software Engineering.

Goal (2) is the concern of data structures and algorithm analysis.

When goal (2) is important, how do we measure an algorithm's cost?

How to Measure Efficiency?

1. Empirical comparison (run programs).

2. Asymptotic Algorithm Analysis.

Critical resources:

Factors affecting running time:

For most algorithms, running time depends on "size" of the input.

Running time is expressed as $\mathbf{T}(n)$ for some function $\mathbf{T}$ on input size n.

Examples of Growth Rate

Example 1:

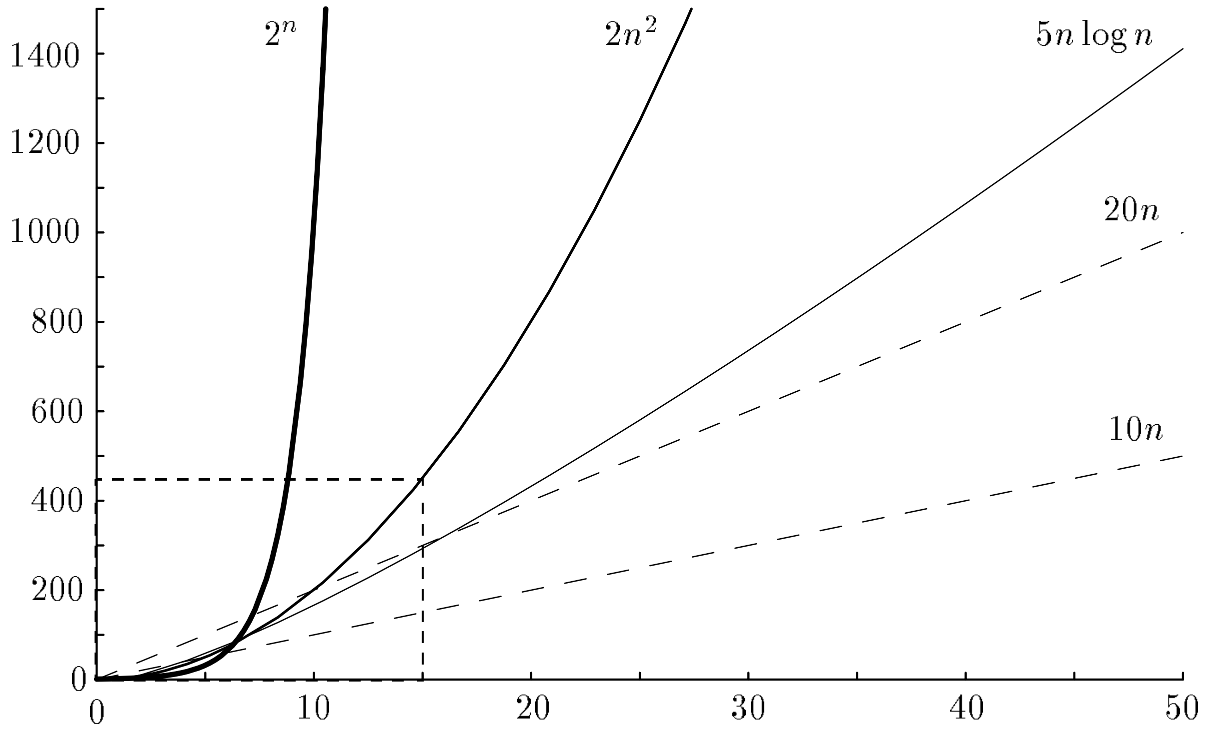
```
static int largest(int[] array) { // Find largest val
  int currLargest = 0;            // Store largest val
  for (int i=0; i<array.length; i++) // For each elem
   if (array[i] > currLargest)    //   if largest
      currLargest = array[i];     //     remember it
  return currLargest;             // Return largest val
}
```

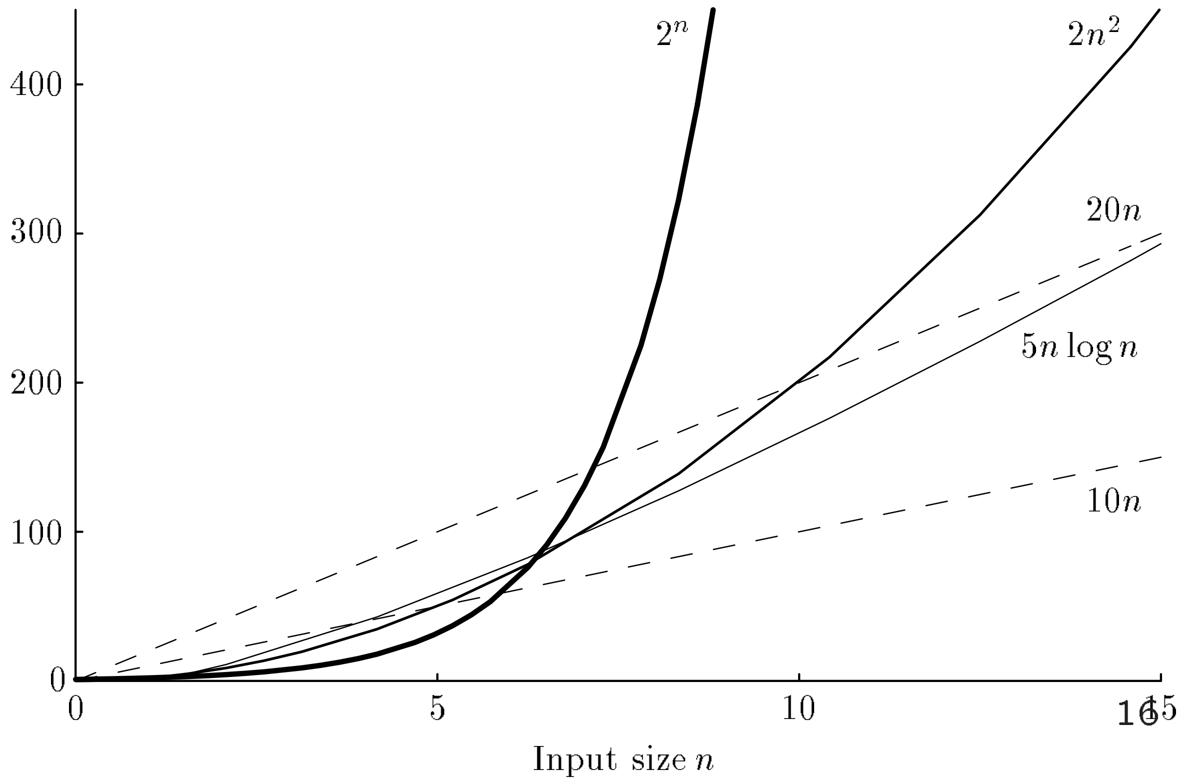
Example 2: Assignment statement

Example 3:

```
sum = 0;
for (i=1; i<=n; i++)
   for (j=1; j<=n; j++)
      sum++;
```

Growth Rate Graph





Best, Worst and Average Cases

Not all inputs of a given size take the same time.

Sequential search for K in an array of n integers:

- Begin at first element in array and look at each element in turn until K is found.

Best Case:

Worst Case:

Average Case:

While average time seems to be the fairest measure, it may be difficult to determine.

When is worst case time important?

Faster Computer or Algorithm?

What happens when we buy a computer 10 times faster?

$\mathbf{T}(n)$	n	n'	Change	n'/n
$10n$	$1,000$	$10,000$	$n' = 10n$	10
$20n$	500	$5,000$	$n' = 10n$	10
$5n \log n$	250	$1,842$	$\sqrt{10}n < n' < 10n$	7.37
$2n^2$	70	223	$n' = \sqrt{10}n$	3.16
2^n	13	16	$n' = n + 3$	$--$

n: Size of input that can be processed in one hour (10,000 steps).

n': Size of input that can be processed in one hour on the new machine (100,000 steps).

Asymptotic Analysis: Big-oh

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\mathrm{O}(f(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \leq cf(n)$ for all $n > n_0$.

Usage: The algorithm is in $\mathrm{O}(n^2)$ in [best, average, worst] case.

Meaning: For *all* data sets big enough (i.e., $n > n_0$), the algorithm *always* executes in less than $cf(n)$ steps [in best, average or worst case].

Upper Bound.

Example: if $\mathbf{T}(n) = 3n^2$ then $\mathbf{T}(n)$ is in $\mathrm{O}(n^2)$.

Wish tightest upper bound:
While $\mathbf{T}(n) = 3n^2$ is in $\mathrm{O}(n^3)$, we prefer $\mathrm{O}(n^2)$.

Big-oh Example

Example 1. Finding value X in an array.

$\mathbf{T}(n) = c_s n/2$.
For all values of $n > 1$, $c_s n/2 \leq c_s n$.
Therefore, by the definition, $\mathbf{T}(n)$ is in O(n) for $n_0 = 1$ and $c = c_s$.

Example 2. $\mathbf{T}(n) = c_1 n^2 + c_2 n$ in average case

$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2 \leq (c_1 + c_2) n^2$ for all $n > 1$.

$\mathbf{T}(n) \leq c n^2$ for $c = c_1 + c_2$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in O(n^2) by the definition.

Example 3: $\mathbf{T}(n) = c$. We say this is in O(1).

Big-Omega

Definition: For $\mathbf{T}(n)$ a non-negatively valued function, $\mathbf{T}(n)$ is in the set $\Omega(g(n))$ if there exist two positive constants c and n_0 such that $\mathbf{T}(n) \geq cg(n)$ for all $n > n_0$.

Meaning: For *all* data sets big enough (i.e., $n > n_0$), the algorithm *always* executes in more than $cg(n)$ steps.

Lower Bound.

Example: $\mathbf{T}(n) = c_1n^2 + c_2n$.

$c_1n^2 + c_2n \geq c_1n^2$ for all $n > 1$.
$\mathbf{T}(n) \geq cn^2$ for $c = c_1$ and $n_0 = 1$.

Therefore, $\mathbf{T}(n)$ is in $\Omega(n^2)$ by the definition.

Want greatest lower bound.

Theta Notation

When big-Oh and Ω meet, we indicate this by using Θ (big-Theta) notation.

Definition: An algorithm is said to be $\Theta(h(n))$ if it is in O($h(n)$) and it is in $\Omega(h(n))$.

Simplifying Rules:

1. If $f(n)$ is in O($g(n)$) and $g(n)$ is in O($h(n)$), then $f(n)$ is in O($h(n)$).
2. If $f(n)$ is in O($kg(n)$) for any constant $k > 0$, then $f(n)$ is in O($g(n)$).
3. If $f_1(n)$ is in O($g_1(n)$) and $f_2(n)$ is in O($g_2(n)$), then $(f_1 + f_2)(n)$ is in O(max($g_1(n)$, $g_2(n)$)).
4. If $f_1(n)$ is in O($g_1(n)$) and $f_2(n)$ is in O($g_2(n)$) then $f_1(n)f_2(n)$ is in O($g_1(n)g_2(n)$).

Running Time of a Program

Example 1: `a = b;`

This assignment takes constant time, so it is $\Theta(1)$.

Example 2:

```
sum = 0;
for (i=1; i<=n; i++)
   sum += n;
```

Example 3:

```
sum = 0;
for (j=1; j<=n; j++)      // First for loop
   for (i=1; i<=j; i++)   //   is a double loop
      sum++;
for (k=0; k<n; k++)       // Second for loop
   A[k] = k;
```

More Examples

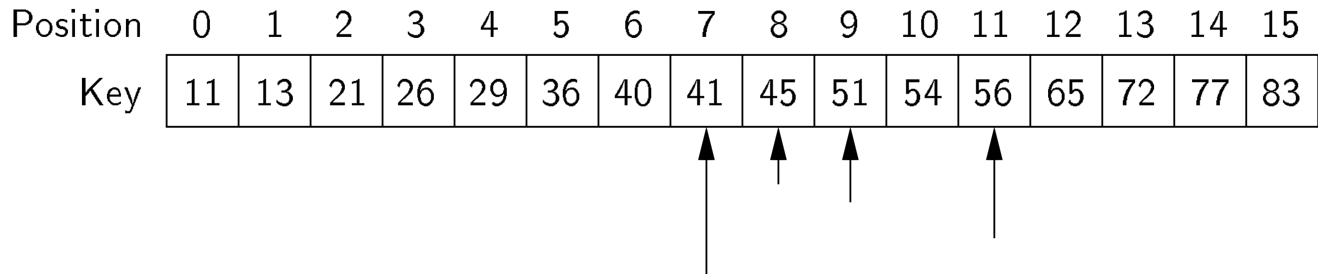
Example 4.

```
sum1 = 0;
for (i=1; i<=n; i++)      // First double loop
   for (j=1; j<=n; j++)   //   do n times
      sum1++;

sum2 = 0;
for (i=1; i<=n; i++)      // Second double loop
   for (j=1; j<=i; j++)   //   do i times
      sum2++;
```

Example 5.

```
sum1 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=n; j++)
      sum1++;

sum2 = 0;
for (k=1; k<=n; k*=2)
   for (j=1; j<=k; j++)
      sum2++;
```

Binary Search




```
static int binary(int K, int[] array,
                         int left, int right) {
  // Return position in array (if any) with value K
  int l = left-1;
  int r = right+1;   // l and r are beyond array bounds
  while (l+1 != r) { // Stop when l and r meet
    int i = (l+r)/2; // Look at middle of subarray
    if (K < array[i]) r = i;     // In left half
    if (K == array[i]) return i; // Found it
    if (K > array[i]) l = i;     // In right half
  }
  return UNSUCCESSFUL; // Search value not in array
}
```

Analysis: How many elements can be examined in the worst case?

Other Control Statements

`while` loop: analyze like a `for` loop.

`if` statement: Take greater complexity of then/else clauses.

`switch` statement: Take complexity of most expensive case.

Subroutine call: Complexity of the subroutine.

Analyzing Problems

Upper bound: Upper bound of best known algorithm.

Lower bound: Lower bound for *every possible algorithm*.

Example: Sorting

1. Cost of I/O: $\Omega(n)$
2. Bubble or insertion sort: $O(n^2)$
3. A better sort (Quicksort, Mergesort, Heapsort, etc.): $O(n \log n)$
4. We prove later that sorting is $\Omega(n \log n)$

Multiple Parameters

Compute the rank ordering for all C pixel values in a picture of P pixels.

```
for (i=0; i<C; i++)   // Initialize count
   count[i] = 0;
for (i=0; i<P; i++)   // Look at all of the pixels
   count[value(i)]++; // Increment proper value count
sort(count);          // Sort pixel value counts
```

If we use P as the measure, then time is $\Theta(P \log P)$.

More accurate is $\Theta(P + C \log C)$.

Space Bounds

Space bounds can also be analyzed with asymptotic complexity analysis.

Time: Algorithm
Space: Data Structure

Space/Time Tradeoff Principle:
One can often achieve a reduction in time is one is willing to sacrifice space, or vice versa.

- Encoding or packing information
  Boolean flags
- Table lookup
  Factorials

Disk Based Space/Time Tradeoff Principle:
The smaller you can make your disk storage requirements, the faster your program will run.