Introduction to Algorithms (Cont.) and Big-oh Notation

Subhabrata Samajder

IIIT, Delhi

Winter Semester,
3rd March, 2023

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.
  - **Elementary operations:** Arithmetic and logical operations.
  - **Finiteness:** It must stop.

Recap: Algorithms

- **Algorithm:** It is a finite sequence of elementary operations with the objective of performing some (computational) task.
- **Input and Output:** Can take *several* inputs but produces a *single* output.
- **Efficiency:** Requiring little 'resources'.
  - **Resources:** Time and space.

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).

  Example: Consider the search problem.
  - $\uparrow$ size of the list $\Rightarrow$ algorithm takes more time.

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).

  Example: Consider the search problem.
  - $\uparrow$ size of the list $\Rightarrow$ algorithm takes more time.

  Example: Is $a \geq 0$ prime?
  - $\uparrow \log_2 a \Rightarrow$ algorithm takes more time.

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).

  Example: Consider the search problem.
  - $\uparrow$ size of the list $\Rightarrow$ algorithm takes more time.

  Example: Is $a \geq 0$ prime?
  - $\uparrow \log_2 a \Rightarrow$ algorithm takes more time.

  Example: $\gcd(a, b)$.
  - $\uparrow n = \log_2(\max\{|a|, |b|\}) \Rightarrow$ algorithm takes more time.

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).

- Thus, one has to factor in the size(s) of the input(s) while talking about algorithmic efficiency.

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).
- Thus, one has to factor in the size(s) of the input(s) while talking about algorithmic efficiency.
- **Note:** Set of all possible inputs is *typically infinite*.

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).
- Thus, one has to factor in the size(s) of the input(s) while talking about algorithmic efficiency.
- **Note:** Set of all possible inputs is *typically infinite*.
- **Size of inputs:** A *function* from the *set of all possible inputs to* $\mathbb{Z}^+$.

Size of input(s)

- **Intuitively:** Time taken by an algorithm will depend on the size(s) of its input(s).
- Thus, one has to factor in the size(s) of the input(s) while talking about algorithmic efficiency.
- **Note:** Set of all possible inputs is *typically infinite*.
- **Size of inputs:** A *function* from the *set of all possible inputs to* $\mathbb{Z}^+$.
- Fixing a positive integer n fixes the set of all inputs of size n and this is a typically a *finite set*.

Size of input(s)

- **Note:** The set of all possible inputs depend on the algorithm and so does the size function.

Size of input(s)

- **Note:** The set of all possible inputs depend on the algorithm and so does the size function.

Example:

- *Search Problem:* $|L|$.
- *Arithmetic Problem:* $\max\{\log_2 a, \log_2 b, \log_2 c, \log_2 d\}$.

Size of input(s)

- **Note:** The set of all possible inputs depend on the algorithm and so does the size function.

Example:

- *Search Problem:* $|L|$.
- *Arithmetic Problem:*
  - Additions: 2
  - Multiplications: 1
  - Time: 2×Cost of Additions + 1×Cost of Multiplication

Runtime Function of an Algorithm

$t(n)$: $\#$ steps required by the algorithm on an input of size n.

Runtime Function of an Algorithm

$t(n)$: # steps required by the algorithm on an input of size n.

Note:

- # steps can vary across two different inputs of size n.

Runtime Function of an Algorithm

$t(n)$: $\#$ steps required by the algorithm on an input of size n.

Note:

- $\#$ steps can vary across two different inputs of size n.
- $\therefore$ given n, one *cannot define a unique $t(n)$* such that the algorithm requires *exactly $t(n)$ steps* on any input of size n.

Two Ways to Tackle this Problem

1. **Worst-case time complexity:** $t(n)$ is the *maximum* of the different numbers of steps that the algorithm requires for different inputs of size n.

Two Ways to Tackle this Problem

1. **Worst-case time complexity:** $t(n)$ is the *maximum* of the different numbers of steps that the algorithm requires for different inputs of size n.
    - May not present a proper picture of the performance.

Two Ways to Tackle this Problem

1. **Worst-case time complexity:** $t(n)$ is the *maximum* of the different numbers of steps that the algorithm requires for different inputs of size n.
   - May not present a proper picture of the performance.
   - May take a rather long time only for a few inputs of size n.

Two Ways to Tackle this Problem

1. **Worst-case time complexity:** $t(n)$ is the *maximum* of the different numbers of steps that the algorithm requires for different inputs of size n.
   - May not present a proper picture of the performance.
   - May take a rather long time only for a few inputs of size n.

   Example: Quick Sort.

Two Ways to Tackle this Problem

1. **Worst-case time complexity:** $t(n)$ is the *maximum* of the different numbers of steps that the algorithm requires for different inputs of size n.
   - May not present a proper picture of the performance.
   - May take a rather long time only for a few inputs of size n.

     Example: Quick Sort.
   - Labelling such an algorithm as inefficient is inappropriate.

Two Ways to Tackle this Problem

1. **Worst-case time complexity:** $t(n)$ is the *maximum* of the different numbers of steps that the algorithm requires for different inputs of size n.
   - May not present a proper picture of the performance.
   - May take a rather long time only for a few inputs of size n.

     Example: Quick Sort.
   - Labelling such an algorithm as inefficient is inappropriate.

2. **Average-case time complexity:** Considers the average case behaviour of the algorithm.
   - For each n, the set of all inputs of size n is assumed to be finite.
   - Define a *uniform distribution* on this set.
   - Then the time function $T(n)$ becomes a *random variable*.
   - *Average-case time complexity* $= E[T(n)]$ (function of n).

Runtime Function of an Algorithm (Cont.)

- We will mostly focus on the worst-case time complexity.

Runtime Function of an Algorithm (Cont.)

- We will mostly focus on the worst-case time complexity.
- Analogously, one can also formulate the worst-case and average-case *space* required by an algorithm.

Arithmetic Problem

$$
\begin{aligned}
&\mathrm{f}(a, b, c, d):\\
&\quad t_1 \;=\; a + b\\
&\quad t_2 \;=\; c + d\\
&\quad t_3 \;=\; t_1 * t_2\\
&\quad \text{return} \quad t_3
\end{aligned}
$$

Arithmetic Problem

$$
\begin{aligned}
&\mathrm{f}(a, b, c, d):\\
&\quad t_1 = a + b\\
&\quad t_2 = c + d\\
&\quad t_3 = t_1 * t_2\\
&\quad \text{return} \quad t_3
\end{aligned}
$$

- **Basic operation:** 2 Addition and 1 multiplication

Arithmetic Problem

$$\begin{aligned} &\mathrm{f}(a, b, c, d): \\ &\quad t_1 = a + b \\ &\quad t_2 = c + d \\ &\quad t_3 = t_1 * t_2 \\ &\quad \text{return} \quad t_3 \end{aligned}$$

- **Basic operation:** 2 Addition and 1 multiplication
  - Depends on the size of integers a, b, c and d.
  - The sizes of a, b, c and d can vary.
  - Assume that $n = \max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil, \lceil \log_2 c \rceil, \lceil \log_2 d \rceil\}$.
  - Adding two n-bit integers take time $\propto n$.
  - Multiplying two n-bit integers take time $\propto n^{\log_2 3}$.

Arithmetic Problem

f(a, b, c, d):

$$
\begin{aligned}
t_1 &= a + b \\
t_2 &= c + d \\
t_3 &= t_1 * t_2 \\
\text{return} \quad & t_3
\end{aligned}
$$

- **Basic operation:** 2 Addition and 1 multiplication
  - Depends on the size of integers a, b, c and d.
  - The sizes of a, b, c and d can vary.
  - Assume that $n = \max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil, \lceil \log_2 c \rceil, \lceil \log_2 d \rceil\}$.
  - Adding two n-bit integers take time $\propto n$.
  - Multiplying two n-bit integers take time $\propto n^{\log_2 3}$.
- **Size of input:** n.

Arithmetic Problem

f(a, b, c, d):

$$
\begin{aligned}
t_1 &= a + b \\
t_2 &= c + d \\
t_3 &= t_1 * t_2 \\
\text{return} \quad & t_3
\end{aligned}
$$

- **Basic operation:** 2 Addition and 1 multiplication
  - Depends on the size of integers a, b, c and d.
  - The sizes of a, b, c and d can vary.
  - Assume that $n = \max\{\lceil \log_2 a \rceil, \lceil \log_2 b \rceil, \lceil \log_2 c \rceil, \lceil \log_2 d \rceil\}$.
  - Adding two n-bit integers take time $\propto n$.
  - Multiplying two n-bit integers take time $\propto n^{\log_2 3}$.
- **Size of input:** n.
- **Time complexity:** $\propto n^{\log_2 3}$.

Searching Problem

- **I/P:** A list L of integer values and another value v.
- **Question:** Does $v \in L$?
- **O/P:** 'index of s' if $v \in L$; else it returns 'FLAG'.

Linear Search

LinearSearch($L[1, \ldots, n], s$):
  for $i = 1$ to n
    if $(L[i] = s)$
      return i;
  endfor;
  return FLAG.

Linear Search

LinearSearch($L[1, \ldots, n], s$):
  for $i = 1$ to n
    if $(L[i] = s)$
      return i;
  endfor;
  return FLAG.

- **Basic operation:** Comparison of the type '$L[i] = s$'

Linear Search

LinearSearch($L[1, \ldots, n], s$):
  for $i = 1$ to n
    if ($L[i] = s$)
      return i;
  endfor;
  return FLAG.

- **Basic operation:** Comparison of the type '$L[i] = s$'
  - Depends on the size of integers $L[i]$'s and s.
  - **Assumption:** The sizes of $L[i]$'s and s are constant.
  - Then, the time taken for each comparison is a *constant*.

Linear Search

LinearSearch($L[1,\ldots,n], s$):
  for $i = 1$ to n
    if ($L[i] = s$)
      return i;
  endfor;
  return FLAG.

- **Basic operation:** Comparison of the type '$L[i] = s$'
  - Depends on the size of integers $L[i]$'s and s.
  - **Assumption:** The sizes of $L[i]$'s and s are constant.
  - Then, the time taken for each comparison is a *constant*.
- **Size of input:** $|L|$.

Linear Search

LinearSearch($L[1,\ldots,n], s$):
  for $i = 1$ to n
    if ($L[i] = s$)
      return i;
  endfor;
  return FLAG.

- **Basic operation:** Comparison of the type '$L[i] = s$'
  - Depends on the size of integers $L[i]$'s and s.
  - **Assumption:** The sizes of $L[i]$'s and s are constant.
  - Then, the time taken for each comparison is a *constant*.
- **Size of input:** $|L|$.
- **Time complexity:** Number of comparisons of the type

$$`L[i] = s'.$$

  - $\#$ other operations $\propto$ $\#$ comparisons.

Linear Search

LinearSearch($L[1, \ldots, n], s$):
  for $i = 1$ to n
    if $(L[i] = s)$
      return i;
  endfor;
  return FLAG.

- **Unsuccessful search:** n comparisons.

Linear Search

LinearSearch($L[1, \ldots, n], s$):
    for $i = 1$ to n
        if $(L[i] = s)$
            return i;
    endfor;
    return FLAG.

- **Unsuccessful search:** n comparisons.
- **Successful search:** Between 1 to n comparisons.

Linear Search

LinearSearch($L[1, \ldots, n], s$):
  for $i = 1$ to n
    if $(L[i] = s)$
      return i;
  endfor;
  return FLAG.

- **Unsuccessful search:** n comparisons.
- **Successful search:** Between 1 to n comparisons.
- **Worst-case complexity:** n comparisons.

Linear Search

LinearSearch($L[1, \ldots, n], s$):
  for $i = 1$ to n
    if $(L[i] = s)$
      return i;
  endfor;
  return FLAG.

- **Unsuccessful search:** n comparisons.
- **Successful search:** Between 1 to n comparisons.
- **Worst-case complexity:** n comparisons.
- $\therefore$ # steps in the worst case $= c_1 n$, for some constant c_1.

Linear Search

Average-case complexity:

- Fix n integers.

Linear Search

Average-case complexity:

- Fix n integers.

- Two cases may arise:
  - **Successful search:** $s \in L[1]$.
    - Call this event succ.
    - Assign $1/n$ (uniform) probability to each of these cases.
    - That is, $\Pr[T(n) = i|\text{succ}] = \frac{(n-1)!}{n!} = \frac{1}{n}, \ \forall \ i = 1, 2, \ldots, n$.

Linear Search

Average-case complexity:

- Fix n integers.

- Two cases may arise:
  - **Successful search:** $s \in L[1]$.
    - Call this event succ.
    - Assign $1/n$ (uniform) probability to each of these cases.
    - That is, $\Pr[T(n) = i|\text{succ}] = \frac{(n-1)!}{n!} = \frac{1}{n}, \ \forall \ i = 1, 2, \ldots, n$.

  - **Unsuccessful search:** $s \notin L$.
    - Call this event unsucc.
    - In this case $\Pr[T(n) = n|\text{unsucc}] = 1$.

Linear Search

Average-case complexity:

- Fix n integers.

- Two cases may arise:
  - **Successful search:** $s \in L[1]$.
    - Call this event succ.
    - Assign $1/n$ (uniform) probability to each of these cases.
    - That is, $\Pr[T(n) = i|\text{succ}] = \frac{(n-1)!}{n!} = \frac{1}{n}, \ \forall\ i = 1, 2, \ldots, n$.

  - **Unsuccessful search:** $s \notin L$.
    - Call this event unsucc.
    - In this case $\Pr[T(n) = n|\text{unsucc}] = 1$.

- Assume that $\Pr[\text{succ}] = \Pr[\text{unsucc}] = \frac{1}{2}$.

Linear Search

Average-case complexity:

- Fix n integers.
- Two cases may arise:
  - **Successful search:** $s \in L[1]$.
    - Call this event succ.
    - Assign $1/n$ (uniform) probability to each of these cases.
    - That is, $\Pr[T(n) = i|\text{succ}] = \frac{(n-1)!}{n!} = \frac{1}{n}$, $\forall\ i = 1, 2, \ldots, n$.
  - **Unsuccessful search:** $s \notin L$.
    - Call this event unsucc.
    - In this case $\Pr[T(n) = n|\text{unsucc}] = 1$.
- Assume that $\Pr[\text{succ}] = \Pr[\text{unsucc}] = \frac{1}{2}$.

Then,

$$
\begin{aligned}
E[T(n)] &= E[T(n)|\text{succ}] \cdot \Pr[\text{succ}] + E[T(n)|\text{unsucc}] \cdot \Pr[\text{unsucc}] \\
&= \frac{1}{2}\left(\sum_{i=1}^{n} i \cdot \frac{1}{n} + n \cdot 1\right) = \frac{3n+1}{4}.
\end{aligned}
$$

Linear Search

Assume: L is sorted in ascending order.

Linear Search

Assume: L is sorted in ascending order.

- **Note:** $\mathrm{LinearSearch}$ cannot take advantage of this information.

Linear Search

Assume: L is sorted in ascending order.

- **Note:** $\mathrm{LinearSearch}$ cannot take advantage of this information.
- **Question:** Can we do better?

Binary Search

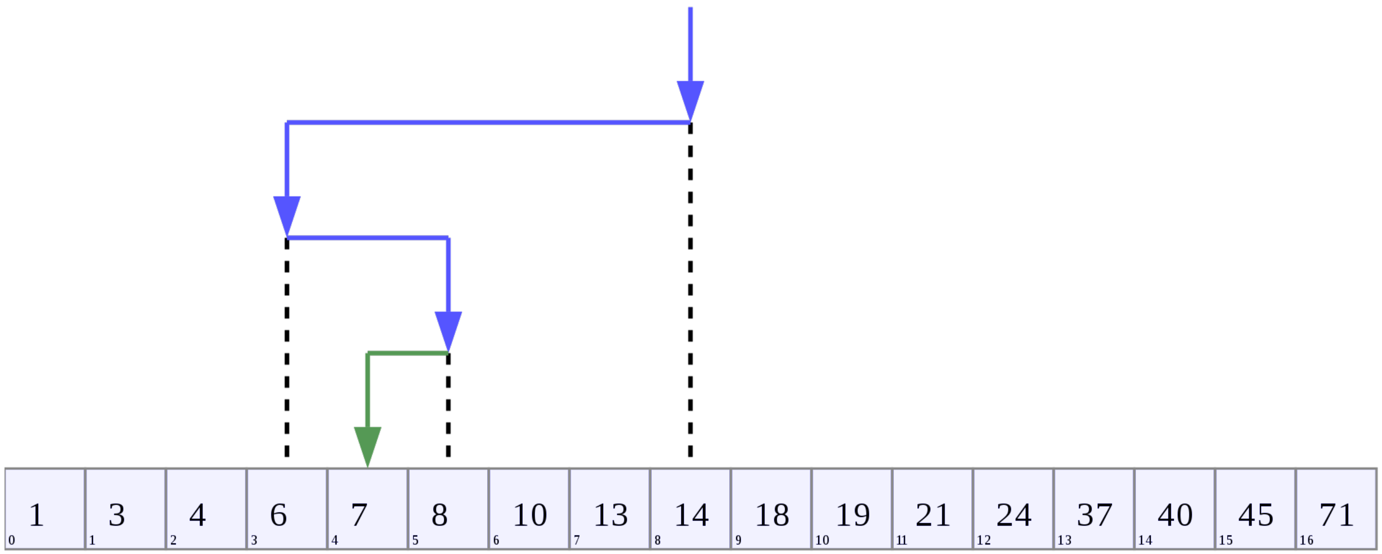

Figure: Binary Search (Courtesy: Wikipedia)

Binary Search (Cont.)

Binary_Search(L, n, s)

I/P: L (a sorted array in the range 1 to n), and z (the search key).

O/P: *Position* (an index i such that $L[i] = s$, or 0 if no such index exist).

Begin

Position := Find($s, 1, n$);

End

Binary Search (Cont.)

Binary_Search(L, n, s)

I/P: L (a sorted array in the range 1 to n), and z (the search key).
O/P: *Position* (an index i such that $L[i] = s$, or 0 if no such index exist).

```
Begin
   Position := Find(s, 1, n);
End

function Find(s, Left, Right) : integer
   Begin
      If (Left = Right)
         If (L[Left] = s)
            return Left;
         else
            return 0;
      else
         Middle := ⌈1/2(Left + Right)⌉;
         If (s < L[Middle])
            return Find(z, Left, Middle − 1);
         else
            return Find(z, Middle, Right);
   End
```

Binary Search (Cont.)

Binary_Search(L, n, s)

I/P: L (a sorted array in the range 1 to n), and z (the search key).
O/P: *Position* (an index i such that $L[i] = s$, or 0 if no such index exist).

```
Begin
   Position := Find(s, 1, n);
End

function Find(s, Left, Right) : integer
   Begin
      If (Left = Right)
         If (L[Left] = s)
            return Left;
         else
            return 0;
      else
         Middle := ⌈1/2(Left + Right)⌉;
         If (s < L[Middle])
            return Find(z, Left, Middle − 1);
         else
            return Find(z, Middle, Right);
   End
```

Homework: Implement BinarySearch in C.

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.

- After each subsequent search the list size becomes $\frac{n}{2^2}, \frac{n}{2^3}, \ldots$.

- **Successful search:** k comparisons, where

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \lceil \log_2 n \rceil.$$

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.
- After each subsequent search the list size becomes $\frac{n}{2^2}, \frac{n}{2^3}, \ldots$.
- **Successful search:** k comparisons, where

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \lceil \log_2 n \rceil.$$

- **Unsuccessful search:** $\lceil \log_2 n \rceil$ comparisons.

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.
- After each subsequent search the list size becomes $\frac{n}{2^2}, \frac{n}{2^3}, \ldots$.
- **Successful search:** k comparisons, where

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \lceil \log_2 n \rceil.$$

- **Unsuccessful search:** $\lceil \log_2 n \rceil$ comparisons.
- **Worst-case time complexity:** $\lceil \log_2 n \rceil$ comparisons.

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.
- After each subsequent search the list size becomes $\frac{n}{2^2}, \frac{n}{2^3}, \ldots$.
- **Successful search:** k comparisons, where

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \lceil \log_2 n \rceil.$$

- **Unsuccessful search:** $\lceil \log_2 n \rceil$ comparisons.
- **Worst-case time complexity:** $\lceil \log_2 n \rceil$ comparisons.
- **Average-case time complexity:** $\lceil \log_2 n \rceil$ comparisons.

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.
- After each subsequent search the list size becomes $\frac{n}{2^2}, \frac{n}{2^3}, \ldots$.
- **Successful search:** k comparisons, where

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \lceil \log_2 n \rceil.$$

- **Unsuccessful search:** $\lceil \log_2 n \rceil$ comparisons.
- **Worst-case time complexity:** $\lceil \log_2 n \rceil$ comparisons.
- $\therefore$ # steps in the worst case $= c_2 \times \lceil \log_2 n \rceil$, $c_2 =$ constant.

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.
- After each subsequent search the list size becomes $\frac{n}{2^2}, \frac{n}{2^3}, \ldots$.
- **Successful search:** k comparisons, where

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \lceil \log_2 n \rceil.$$

- **Unsuccessful search:** $\lceil \log_2 n \rceil$ comparisons.
- **Worst-case time complexity:** $\lceil \log_2 n \rceil$ comparisons.
- $\therefore$ # steps in the worst case $= c_2 \times \lceil \log_2 n \rceil$, $c_2 =$ constant.
- Clearly, BinarySearch is "better" than LinearSearch.

Binary Search (Cont.)

- After one search the size of the search space is reduced by *half*, i.e., becomes $n/2$.
- After each subsequent search the list size becomes $\frac{n}{2^2}, \frac{n}{2^3}, \ldots$.
- **Successful search:** k comparisons, where

$$\frac{n}{2^k} = 1 \quad \Rightarrow \quad k = \lceil \log_2 n \rceil .$$

- **Unsuccessful search:** $\lceil \log_2 n \rceil$ comparisons.
- **Worst-case time complexity:** $\lceil \log_2 n \rceil$ comparisons.
- $\therefore$ # steps in the worst case $= c_2 \times \lceil \log_2 n \rceil$, $c_2 =$ constant.
- Clearly, BinarySearch is "better" than LinearSearch.
- **Question:** Which is the "best" possible algorithm for a given 'problem'?

Comparing Algorithms

Consider a problem Π.

- Any algorithm which solves Π will take as input an instance of the problem and return the correct answer.

Comparing Algorithms

Consider a problem Π.

- Any algorithm which solves Π will take as input an instance of the problem and return the correct answer.
- Two algorithms for the same problem can be compared by comparing their time complexities.
- More generally, one can ask for the best possible algorithm to solve Π or to show that Π cannot be solved efficiently.

Comparing Algorithms

Consider a problem Π.

- Any algorithm which solves Π will take as input an instance of the problem and return the correct answer.
- Two algorithms for the same problem can be compared by comparing their time complexities.
- More generally, one can ask for the best possible algorithm to solve Π or to show that Π cannot be solved efficiently.
- Answering such questions form the motivation for the rich area of algorithm design and analysis (ADA).

Asymptotic Notation

- **Measure of performance:** Worst-case complexity.

Asymptotic Notation

- **Measure of performance:** Worst-case complexity.
  - LinearSearch: $t(n) = c_1 n$.
  - BinarySearch: $t(n) = c_2 \times \lceil \log_2 n \rceil$.

Asymptotic Notation

- **Measure of performance:** Worst-case complexity.
  - LinearSearch: $t(n) = c_1 n$.
  - BinarySearch: $t(n) = c_2 \times \lceil \log_2 n \rceil$.
  - **Note:** The constants c_1 and c_2 depends upon many things including implementation details.

Asymptotic Notation

- **Measure of performance:** Worst-case complexity.
  - LinearSearch: $t(n) = c_1 n$.
  - BinarySearch: $t(n) = c_2 \times \lceil \log_2 n \rceil$.
  - **Note:** The constants c_1 and c_2 depends upon many things including implementation details.
  - Would be convenient to have a method which does not involve these constants.

Big-oh Notation

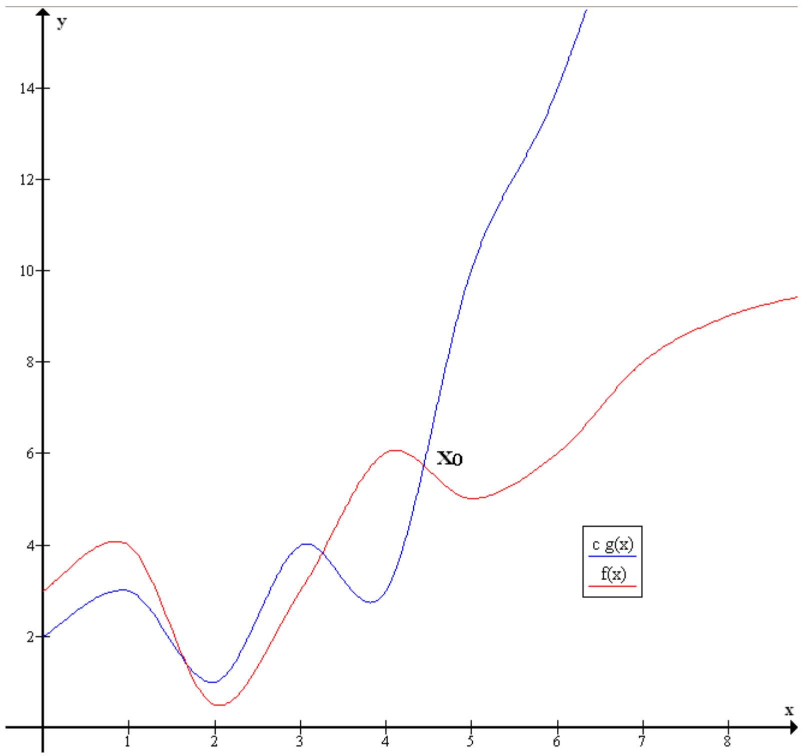

Figure: Binary Search (Courtesy: Wikipedia)

Big-oh Notation (Cont.)

Definition ($\mathcal{O}$-notation)

Let g and f be functions from the set of natural numbers to itself. The function f is said to be $\mathcal{O}(g)$ **(read big-oh of g)**, if there is a constant c and a natural n_0 such that

$$f(n) \leq cg(n) \text{ for all } n > n_0.$$

Big-oh Notation (Cont.)

Definition ($\mathcal{O}$-notation)

Let g and f be functions from the set of natural numbers to itself. The function f is said to be $\mathcal{O}(g)$ **(read big-oh of g)**, if there is a constant c and a natural n_0 such that

$$f(n) \leq cg(n) \text{ for all } n > n_0.$$

- LinearSearch $= \mathcal{O}(n)$ (both cases)
- BinarySearch $= \mathcal{O}(\log n)$ (both cases)
  - **Homework:** Derive the average-case complexity of BinarySearch with early termination.

Big-oh Notation (Cont.)

Definition ($\mathcal{O}$-notation)

Let g and f be functions from the set of natural numbers to itself. The function f is said to be $\mathcal{O}(g)$ **(read big-oh of g)**, if there is a constant c and a natural n_0 such that

$$f(n) \leq cg(n) \text{ for all } n > n_0.$$

- LinearSearch $= \mathcal{O}(n)$ (both cases)
- BinarySearch $= \mathcal{O}(\log n)$ (both cases)
  - **Homework:** Derive the average-case complexity of BinarySearch with early termination.
- **Caveat:**
  1. We lose a lot of details.
  2. Details can be important in actual practice.

Big-oh Notation (Cont.)

Note: $\mathcal{O}(g)$ is a set!

Big-oh Notation (Cont.)

Note: $\mathcal{O}(g)$ is a set!

Abuse of notation: We write $f = \mathcal{O}(g)$ to mean $f \in \mathcal{O}(g)$.

Big-oh Notation (Cont.)

Note: $\mathcal{O}(g)$ is a set!

Abuse of notation: We write $f = \mathcal{O}(g)$ to mean $f \in \mathcal{O}(g)$.

Examples:

- $5n^2 + 15 = \mathcal{O}(n^2)$ [$\because$ $5n^2 + 15 \leq 6n^2$ for $n > 4$].

Big-oh Notation (Cont.)

Note: $\mathcal{O}(g)$ is a set!

Abuse of notation: We write $f = \mathcal{O}(g)$ to mean $f \in \mathcal{O}(g)$.

Examples:

- $5n^2 + 15 = \mathcal{O}(n^2)$ [$\because$ $5n^2 + 15 \leq 6n^2$ for $n > 4$].
- $5n^2 + 15 = \mathcal{O}(n^3)$ [$\because$ $5n^2 + 15 \leq n^3$ for all $n > 6$].

Big-oh Notation (Cont.)

Note: $\mathcal{O}(g)$ is a set!

Abuse of notation: We write $f = \mathcal{O}(g)$ to mean $f \in \mathcal{O}(g)$.

Examples:

- $5n^2 + 15 = \mathcal{O}(n^2)$ [$\because$ $5n^2 + 15 \leq 6n^2$ for $n > 4$].
- $5n^2 + 15 = \mathcal{O}(n^3)$ [$\because$ $5n^2 + 15 \leq n^3$ for all $n > 6$].
- $\mathcal{O}(1)$ denote a constant.

Big-oh Notation (Cont.)

Note: $\mathcal{O}(g)$ is a set!

Abuse of notation: We write $f = \mathcal{O}(g)$ to mean $f \in \mathcal{O}(g)$.

Examples:

- $5n^2 + 15 = \mathcal{O}(n^2)$ [$\because$ $5n^2 + 15 \leq 6n^2$ for $n > 4$].
- $5n^2 + 15 = \mathcal{O}(n^3)$ [$\because$ $5n^2 + 15 \leq n^3$ for all $n > 6$].
- $\mathcal{O}(1)$ denote a constant.
- One can include constants within the $\mathcal{O}$ notation.
- But there is no reason to do it.
- We therefore write $\mathcal{O}(n)$ instead of $\mathcal{O}(5n + 4)$.

Poly-time vs. Exponential Algorithm

Poly-time:

- $\mathcal{O}(n^c)$.
- Identified with *efficient* algorithms.

Poly-time vs. Exponential Algorithm

Poly-time:

- $\mathcal{O}(n^c)$.
- Identified with *efficient* algorithms.

Exponential-time:

- $\mathcal{O}(2^n)$.
- Identified with *inefficient* algorithms.

Poly-time vs. Exponential Algorithm

Poly-time:

- $\mathcal{O}(n^c)$.
- Identified with *efficient* algorithms.

Exponential-time:

- $\mathcal{O}(2^n)$.
- Identified with *inefficient* algorithms.

For any given problem, it is of interest to be able to design a polynomial time algorithm to solve it.

Some Results

Monotonically growing function: If $n_1 > n_2 \Rightarrow f(n_1) \geq f(n_2)$.

Some Results

Monotonically growing function: If $n_1 > n_2 \;\Rightarrow\; f(n_1) \geq f(n_2)$.

Theorem

For all constants $c > 0$ and $a > 1$, and for all monotonically growing functions $f(n)$,

$$(f(n))^c = \mathcal{O}(a^{f(n)}).$$

In other words, an exponential function grows faster than does a polynomial function.

Some Results

Monotonically growing function: If $n_1 > n_2 \;\Rightarrow\; f(n_1) \geq f(n_2)$.

Theorem

For all constants $c > 0$ and $a > 1$, and for all monotonically growing functions $f(n)$,

$$(f(n))^c = \mathcal{O}(a^{f(n)}).$$

In other words, an exponential function grows faster than does a polynomial function.

Proof: Home Work!

Some Results

Monotonically growing function: If $n_1 > n_2 \;\Rightarrow\; f(n_1) \geq f(n_2)$.

Theorem

For all constants $c > 0$ and $a > 1$, and for all monotonically growing functions $f(n)$,

$$(f(n))^c = \mathcal{O}(a^{f(n)}).$$

In other words, an exponential function grows faster than does a polynomial function.

Proof: Home Work!

Corollaries:

- Putting $f(n) = n$, we get $n^c = \mathcal{O}(a^n)$.
- Putting $f(n) = \log_a n$, we get $(\log_a n)^c = \mathcal{O}(a^{\log_a n}) = \mathcal{O}(n)$.

Some Results (Cont.)

Lemma

1. *If* $f(n) = \mathcal{O}(s(n))$ *and* $g(n) = \mathcal{O}(r(n))$ *then*
$$f(n) + g(n) = \mathcal{O}(s(n) + r(n)).$$
2. *If* $f(n) = \mathcal{O}(s(n))$ *and* $g(n) = \mathcal{O}(r(n))$ *then*
$$f(n).g(n) = \mathcal{O}(s(n).r(n)).$$

Some Results (Cont.)

Lemma

1. *If* $f(n) = \mathcal{O}(s(n))$ *and* $g(n) = \mathcal{O}(r(n))$ *then*
$$f(n) + g(n) = \mathcal{O}(s(n) + r(n)).$$
2. *If* $f(n) = \mathcal{O}(s(n))$ *and* $g(n) = \mathcal{O}(r(n))$ *then*
$$f(n).g(n) = \mathcal{O}(s(n).r(n)).$$

Proof: Home Work!

Some Results (Cont.)

Note:

- It is not possible to *subtract* or *divide*.

Some Results (Cont.)

Note:

- It is not possible to *subtract* or *divide*.
- That is, it is not true in general that $f(n) = \mathcal{O}(s(n))$ and $g(n) = \mathcal{O}(r(n))$ imply that

$$f(n) - g(n) = \mathcal{O}(s(n) - r(n))$$

or that

$$f(n)/g(n) = \mathcal{O}(s(n)/r(n)).$$

Some Results (Cont.)

Note:

- It is not possible to *subtract* or *divide*.
- That is, it is not true in general that $f(n) = \mathcal{O}(s(n))$ and $g(n) = \mathcal{O}(r(n))$ imply that

$$f(n) - g(n) = \mathcal{O}(s(n) - r(n))$$

or that

$$f(n)/g(n) = \mathcal{O}(s(n)/r(n)).$$

(Show it!)

Better Processors vs. Efficient Algorithms

running times	$time_1$ 1000 steps/sec	$time_2$ 2000 steps/sec	$time_3$ 4000 steps/sec	$time_4$ 8000 steps/sec
$\log_2 n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n \log_2 n$	10	5	2.5	1.25
$n^{1.25}$	32	16	8	4
n^2	1,000	500	250	125
n^3	1,000,000	500,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Table: Running times (in seconds) under different assumptions ($n = 1000$).

Books Consulted

1. Chapter 2 of *A Course on Cooperative Game Theory* by Satya R. Chakravarty, Palash Sarkar and Manipushpak Mitra.

2. *Introduction to Algorithms: A Creative Approach* by Udi Manber.

Thank You for your kind attention!