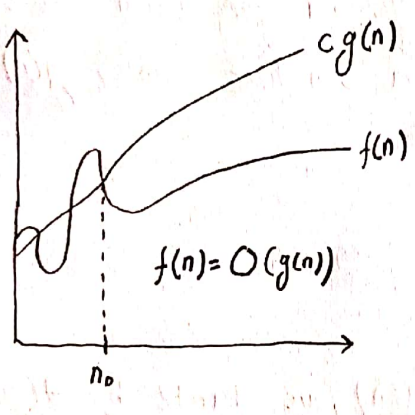
Algorithm Analysis

Asymptotic Notation & Functions & Running Times

We use asymptotic notation primarily to describe the running times of algorithms. Asymptotic notation actually applies to functions, however. The functions to which we apply asymptotic notation will usually characterize the running times of algorithms. Even when we use asymptotic notation to apply to the running time of an algorithm, we need to understand which running time we mean.

O-Notation



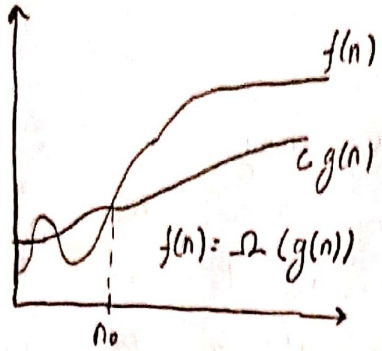
When we have only an asymptotic upper bound, we use O-notation. For a given function $g(n)$, we denote by $O(g(n))$ the set of functions.

$O(g(n)) = \{ f(n) :$ there exist positive constants c and n_0 such that $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0 \}$

Since O-notation describes an upper bound, 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.

- if $f(n)$ is a polynomial of degree d then $f(n)$ is $O(n^d)$
- Use the smallest possible class of functions
  say "$2n$ is $O(n)$" instead of "$2n$ is $O(n^2)$"
- if $d(n)$ is $O(f(n))$ and $e(n)$ is $O(g(n))$ then
  - $d(n) + e(n)$ is $O(f(n) + g(n))$
  - $d(n) \cdot e(n)$ is $O(f(n) \cdot g(n))$
- if $d(n)$ is $O(g(n))$ and $g(n)$ is $O(f(n))$ then $d(n)$ is $O(g(n))$
- if $p(n)$ is a polynomial in n then $\log p(n)$ is $O(\log n)$

Ω - Notation

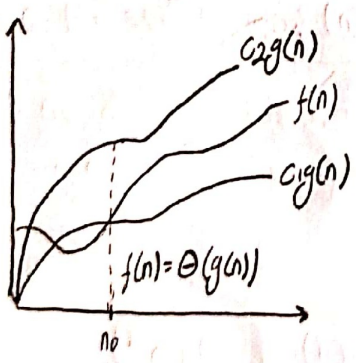


Ω - notation (omega) provides an asymptotic lower bound. For a given function $g(n)$, we denote by $\Omega(g(n))$ the set of functions:

$\Omega(g(n)) = \{ f(n) :$ there exists positive constants c and n_0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n_0 \}$

When we say that the running time of an algorithm is $\Omega(g(n))$, we mean that no matter what particular input of size n is chosen for each value of n, the running time on that input is at least a constant times $g(n)$, for sufficiently large n_0.

Θ - Notation



For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions.

$\Theta(g(n)) = \{ f(n) :$ there exist positive constants c_1, c_2, n_0 such that $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n \geq n_0 \}$

We say that $g(n)$ is an asymptotically tight bound for $f(n)$.

Examples

- $2n+10$ is $O(n)$

  $\rightarrow 2n+10 \leq cn$

  $(c-2)n \geq 10$

  $n \geq 10/c-2$

  pick $c=3$ and $n_0 = 10$

- n^2 is not $O(n)$

  $\rightarrow n^2 \leq cn$

  $n \leq c$

  This inequality cannot be satisfied since c is a constant.

- $7n-2$ is $O(n)$

  $\rightarrow 7n-2 \leq cn$

  $(7-c)n \leq 2$

  pick $c=7$ and $n_0 = 1$

- $3n^3+20n^2+5$ is $O(n^3)$

  $\rightarrow 3n^3+20n^2+5 \leq cn^3$

  $20n^2+5 \leq (c-3)n^3$

  pick $c=4$, $n_0 = 21$

- $5n^2$ is $\Omega(n^2)$

  $\rightarrow cn^2 \leq 5n^2$

  $c \leq 5$

  pick $c=5$, $n_0 = 1$

- $5n^2$ is $\Omega(n)$

  $\rightarrow cn \leq 5n^2$

  $c \leq 5n$

  pick $c=1$, $n_0 = 1$

- $5n^2$ is $\Theta(n^2)$

  $\rightarrow c_1 n^2 \leq 5n^2 \leq c_2 n^2$

  $c_1, c_2 = 5$ and $n_0 = 1$

- $4n^2+2n$ is $O(n)$

  $\rightarrow 4n^2+2n \leq cn$

  $4n+2 \leq c$

  $n \leq (c-2)/4$

  Since c is constant inequality cannot be achieved.

- $3n^2-100n+6 = O(n^2) \rightarrow c=3 \rightarrow 3n^2 > 3n^2-100n+6$
- $3n^2-100n+6 = O(n^3) \rightarrow c=1 \rightarrow n^3 > 3n^2-100n+6 \longrightarrow n_0 = 4$
- $3n^2-100n+6 \neq O(n) \longrightarrow cn < 3n^2$ when $n > c$
- $3n^2-100n+6 = \Omega(n^2) \rightarrow c=2 \rightarrow 2n^2 < 3n^2-100n+6 \longrightarrow n_0 = 101$
- $3n^2-100n+6 \neq \Omega(n^3) \rightarrow c=1 \rightarrow n^3 > 3n^2-100n+6 \longrightarrow n > 3$
- $3n^2-100n+6 = \Omega(n) \rightarrow cn < 3n^2-100n+6 \longrightarrow n > 100c$

- $3n^2-100n+6 = \Theta(n^2)$, because both O and Ω apply.
- $3n^2-100n+6 \neq \Theta(n^3)$, because only O applies.
- $3n^2-100n+6 \neq \Theta(n)$, because only Ω applies.

- $f(n) = 2n+10$, $g(n) = n$ $\Rightarrow$ $f(n) \in O(g(n))$, $g(n) \in O(f(n))$

$\rightarrow 2n+10 \leq cn$

$10 \leq (c-2)n$

$10/(c-2) \leq n$

$c=3, n_0=10$

$\rightarrow n \leq c \cdot (2n+10)$

$n \leq 2nc+10c$

$(1-2c)n \leq 10c$

$c=1, n_0=0$

- $2n+5$ is $\Theta(n)$

$\rightarrow 2n+5 \geq c_1 n$

$5 \geq (c_1-2)n$

$c_1=2, n_0=1$

$\rightarrow 2n+5 \leq c_2 n$

$5 \leq (c_2-2)n$

$c_2=7, n_0=1$

- 2^{n+1} is $\Theta(2^n)$

$\rightarrow 2^{n+1} \leq c_1 2^n$

$2 \cdot 2^n \leq c_1 2^n$

$c_1=2, n_0=1$

$\rightarrow c_2 2^n \leq 2^{n+1}$

$c_2 2^n \leq 2 \cdot 2^n$

$c_2=2, n_0=1$

- $(x+y)^2$ is $O(x^2+y^2)$

$\rightarrow (x+y)^2 \leq n(x^2+y^2)$

$x^2+2xy+y^2 \leq nx^2+ny^2$ if no $2xy$ then inequality holds $c \geq 1$

if $x \leq y$ then $2xy \leq 2y^2 \leq 2(x^2+y^2)$

if $y \leq x$ then $2xy \leq 2x^2 \leq 2(x^2+y^2)$

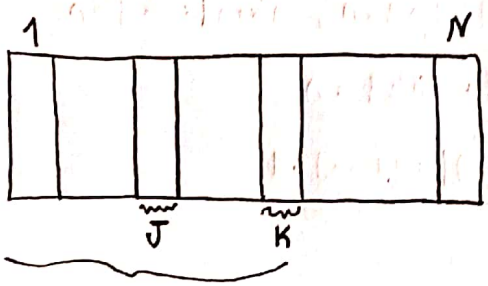
Thus $c=2+1=3$

$(x+y)^2 \leq 3(x^2+y^2)$

Insetion Sort Average Case

Worst case $\longrightarrow O(n^2), \Omega(n^2) \longrightarrow \Theta(n^2)$

best case $\longrightarrow O(n), \Omega(n) \longrightarrow \Theta(n)$



Number of operations in inner while

$$\frac{1}{K} \cdot \sum_{J=1}^{K} (K-J+1)$$

$$= \frac{1}{K} \cdot \left(K^2 - \frac{K \cdot (K+1)}{2} + K\right)$$

$$= \frac{2K^2 - K^2 - K + 2K}{2K} = \frac{K+1}{2}$$

$$\sum_{K=2}^{N} \frac{K+1}{2} = \frac{1}{2} \cdot \frac{(N+1) \cdot (N+2)}{2} \longrightarrow \text{Outer loop}$$

- Average Case $= \frac{1}{4}\left[N^2+3N+2\right] \longrightarrow O(n^2)$

Using Limit For Comparing Order Of Growths

$$\lim_{n\to\infty} \frac{t(n)}{g(n)} = \begin{cases} 0, \text{ implies that } t(n) \text{ has a smaller order of growth than } g(n) \\ c \in \mathbb{R}, \text{ implies that } t(n) \text{ has same order of growth as } g(n) \\ \infty, \text{ implies that } t(n) \text{ has a larger order of growth than } g(n). \end{cases}$$

- $t(n) = \frac{1}{2}n(n-1)$, $g(n) = n^2$

$$\lim_{n\to\infty} \frac{\frac{1}{2} n \cdot (n-1)}{n^2} = \frac{1}{2} \lim_{n\to\infty} \frac{n^2-n}{n^2}$$

$$= \frac{1}{2} \lim_{n\to\infty} \left(1 - \frac{1}{n}\right) = 1/2$$

Useful Formulas For The Analysis Of Algorithms

Logarithms

- $\log_x y = y^{\log x}$
- $\log xy = \log x + \log y$
- $\log x/y = \log x - \log y$
- $\log_a x = \log_a b \log_b x$
- $a^{\log_b x} = x^{\log_b a}$

Floor And Ceiling

- $x-1 < \lfloor x \rfloor \leq x \leq \lceil x \rceil < x+1$
- $\lfloor x+n \rfloor = \lfloor x \rfloor + n$, $\lceil x+n \rceil = \lceil x \rceil + n$
- $\lfloor n/2 \rfloor + \lceil n/2 \rceil = n$
- $\lceil \log(n+1) \rceil = \lfloor \log n \rfloor + 1$

Summations

* $\sum_{i=L}^{U} 1 = U - L + 1$
* $\sum_{i=1}^{n} i = \frac{n \cdot (n+1)}{2}$
* $\sum_{i=1}^{n} i^K \approx \frac{1}{K+1} \cdot n^{K+1}$
* $\sum_{i=1}^{n} 1 = n$
* $\sum_{i=1}^{n} i^2 = \frac{n \cdot (n+1) \cdot (2n+1)}{6}$
* $\sum_{i=1}^{n} a^i = \frac{a^{n+1} - 1}{a - 1}$
* $\sum_{i=1}^{n} i 2^i = (n-1)2^{n+1} + 2$
* $\sum_{i=1}^{n} \frac{1}{i} \approx \ln n + \gamma$, $\gamma \approx 0.5772$
* $\sum_{i=L}^{U} c a_i = c \cdot \sum_{i=L}^{U} a_i$
* $\sum_{i=L}^{U} a_i \pm b_i = \sum_{i=L}^{U} a_i \pm \sum_{i=L}^{U} b_i$

Mathematical Analysis Of Non-Recursive Algorithms

```
MaxElement(A[0..n-1])
    max ← A[0]
    for i ← 1 to n-1 do
        if A[i] > max
            max ← A[i]
    return max
```

→ Comparison: $A[i] < max$ ①

→ Assignment: $max \leftarrow A[i]$ ②

$$C(n) = \sum_{i=1}^{n-1} 1 = n-1 \in O(n)$$, where n is number of elements.

↓

$C_{worst}(n)$ or $C_{avg}(n)$ or $C_{best}(n)$

```
UniqueElements(A[0..n-1])
    for i ← 0 to n-2 do
        for j ← i+1 to n-1 do
            if A[i] = A[j]
                return false
    return true
```

$$C_{worst}(n) = \sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} 1 = \sum_{i=1}^{n-2} \left(\sum_{j=i+1}^{n-1} 1 \right)$$

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

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

$$= \frac{n \cdot (n-1)}{2} \approx \frac{1}{2} n^2 \in \Theta(n^2)$$

$$T(n) \approx c \cdot C(n) = c \cdot n^2$$

→ cost of basic operations

→ Running time of algorithm

Mathematical Analysis Of Recursive Algorithms

```
factorial (n)
    if n=0
        return 1
    else
        return n * factorial(n-1)
```

$F(n) = F(n-1) \cdot n$

$M(n) = M(n-1) + 1$

- $M(n-1)$ → to compute $F(n-1)$
- $+1$ → to multiply $F(n-1)$ by n

Recurrence Relation ⟶ Initial Condition: if $n=0$ return 1 ⟶ $M(0) = 0$ (no multiplication)

$M(n) = M(n-1) + 1$ for $n > 0$

$M(0) = 0$

The calls stop when $n=0$

Method Of Backward Substitution:

$M(n) = M(n-1) + 1$, substitute $M(n-1) = M(n-2) + 1$

$= [M(n-2) + 1] + 1 = M(n-2) + 2$, substitute $M(n-2) = M(n-3) + 1$

$= [M(n-3) + 1] + 2 = M(n-3) + 3$,

⋮

$= M(n) = M(n-i) + i$

⟶ $i \leftarrow n$

$M(n-n) + n$

$= M(0) + n$

$= n$ //

```
binary Digit (n)
    if n = 1
        return 1
    else
        return binary Digit (⌊n/2⌋)
```

→ Addition: $A(n) = A(\lfloor n/2 \rfloor) + 1$ for $n > 1$

→ Initial Condition: $A(1) = 0$

Smoothness rule ⟶ $n = 2^k$

$$A(2^k) = A(2^{k-1}) + 1 \quad \text{for } n > 1$$

$$A(2^0) = A(1) = 0$$

↳ Backward:

$$A(2^k) = A(2^{k-1}) + 1$$

$$= A(2^{k-2}) + 2$$

$$= A(2^{k-3}) + 3$$

$$\vdots$$

$$= A(2^{k-c^0}) + c^0$$

$\hookrightarrow c^0 \leftarrow k$

$$A(2^0) + k = 0 + k = \log_2 n \in \Theta(\log n)$$

- $X(n) = X(n-1) + n, \quad X(0) = 0$

$$= [X(n-2) + n-1] + n = X(n-2) + (n-1) + n$$

$$= [X(n-3) + n-2] + (n-1) + n = X(n-3) + (n-2) + (n-1) + n$$

$$\vdots$$

$$= X(n-c^0) + (n-c^0+1) + (n-c^0+2) + \cdots + n$$

$c^0 \leftarrow n$

$$\underbrace{X(0)}_{0} + 1 + 2 + \cdots + n = \frac{n \cdot (n+2)}{2} \in \Theta(n^2)$$

- $X(n) = X(n/3) + 1 \quad , X(1) = 1$

$n = 3^K$

$X(3^K) = X(3^{K-1}) + 1$

$= X(3^{K-2}) + 2$

$\vdots$

$= X(3^{K-c^\circ}) + c^\circ$

$\longrightarrow \quad c^\circ \leftarrow K$

$X(1) + K = K + 1 = \log_3 n + 1 \in \Theta(\log n)$

- $\sum_{c^\circ=2}^{n-1} \log_2 c^{\circ 2}$, find the order of growth

$= 2 \cdot \sum_{c^\circ=2}^{n-1} \log_2 c^\circ = 2 \cdot \sum_{c^\circ=1}^{n} \log_2 c^\circ - 2 \cdot \log_2 n$

$2\Theta(n \log n) - 2\Theta(\log n) = \Theta(n \log n)$

Brute Force

Brute-force is a straight-forward approach to solving a problem, usually directly based on the problem statement and definitions of the concepts.

- $a^n \rightarrow \underbrace{a * a * \ldots * a}_{n \text{ times}}$
- Selection Sort and Bubble Sort.
- Linear (Sequential) search
- Brute Force String Matching
- Convex-Hull
- Closest Pair

Exhaustive Search

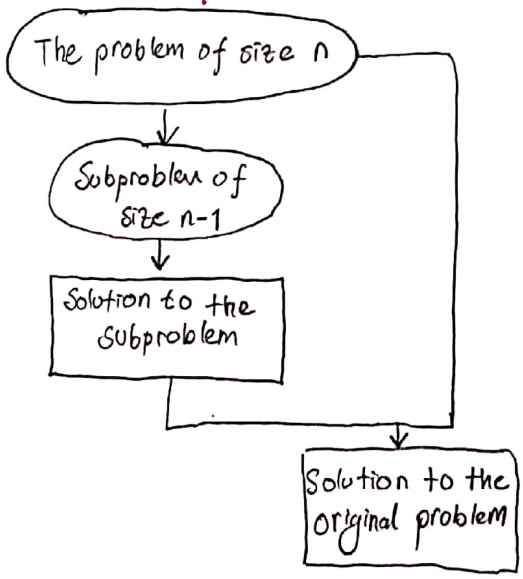
Exhaustive Search is simply a brute-force approach to Combinatorial problems.

- Traveling Salesman Problem
- Knapstack Problem

Decrease-and-Conquer

- Decrease by a constant.
- Decrease by a constant factor.
- Variable size decrease.

Decrease-by-one-and-Conquer



$$a^n = a^{n-1} \cdot a$$

$$f(n) = \begin{cases} f(n-1) * a & \text{if } n > 0 \\ 1 & \text{if } n = 0 \end{cases}$$

Decrease-by-a-constant Factor

$$a^n = \begin{cases} (a^{n/2})^2 & \text{if } n \text{ is even} \\ (a^{(n-1)/2})^2 * a & \text{if } n \text{ is odd} \\ 1 & \text{if } n = 0 \end{cases}$$

Variable-size-decrease

$$gcd(m,n) = gcd(n, m \bmod n)$$

Binary Search Algorithm

- Decrease-by-constant-factor

```
L ← 0
R ← n-1
while L <= R do
    m ← ⌊(L+R)/2⌋
    if K = A[m]          → comparison every time
        return m
    else if K < A[m]
        R ← m-1
    else
        L ← m+1
return -1
```

$$C_w(n) = C_w(\lfloor n/2 \rfloor) + 1 \text{ for } n > 1, \quad C_w(1) = 1$$

$$C_w(2^k) = k+1 = \log_2 n + 1$$

Divide and Conquer

In divide-and-conquer, we solve a problem recursively, applying three steps at each level of the recursion.

- Divide the problem into a number of subproblems that are smaller instances of the same problem
- Conquer the subproblems by solving them recursively. If the problem sizes are small enough, however, just solve the subproblems in a straightforward manner.
- Combine the solutions to the subproblems into the solution for the original problem.

The Master Theorem

Let $a \in \mathbb{N}^{\geq 1}$ and $b \in \mathbb{N}^{>1}$ be constants, let $f(n)$ be a function, and let $T(n)$ be defined on the non-negative integers by the recurrence.

$$T(n) = aT(n/b) + f(n) \quad ; \quad f(n) \in \Theta(n^d)$$

Then $T(n)$ has the asymptotic bounds:

- $T(n) \in \Theta(n^d)$, if $a < b^d$
- $T(n) \in \Theta(n^d \log n)$, if $a = b^d$
- $T(n) \in \Theta(n^{\log_b a})$, if $a > b^d$

$$T(n) = a \cdot T(n/b) + f(n)$$

→ Our division of the problem yields a subproblems, each of which is $1/b$ size of the original

For merge sort $D(n) \in \Theta(1)$

$C(n) \in \Theta(n)$

→ $f(n) = D(n) + C(n)$

running time of Divide and Conquer

- $A(n) = 2A(n/2) + 1$

$a = 2$

$b = 2$

$f(n) \in \Theta(n^d)$

$n^d = 1, \ d = 0$

$$A(n) \in \Theta(n^{\log_b a}) = \Theta(n^{\log_2 2}) = \Theta(n)$$

Merge Sort vs Quick Sort

Merge Sort Complexity : $\Theta(n \log n)$ ⟶ Best case
Average case
Worst case

Quick Sort Complexity: $\Theta(n \log n)$ ⟶ best case
$\Theta(n^2)$ ⟶ worst case
$\Theta(n \log n)$ ⟶ Average case

Merge Sort

```
MergeSort(A[0..n-1], p, r)
    if (p < r)
        q ← (p+r)/2
        MergeSort(A, p, q)
        MergeSort(A, q+1, r)
        Merge(A, p, q, r)
```

```
① for c ← p to r do
        A[c] ← B[c-p]
```

```
Merge(A[0..n-1], p, q, r)
    nB ← r-p+1
    let B is B[0..nB]
    c ← p
    J ← q+1
    K ← 0
    while ((c ≤ q) and (J ≤ r)) do
        if A[c] < A[J]
            B[K] ← A[c]
            c ← c+1
        else
            B[K] ← A[J]
            J ← J+1
        K ← K+1
    while c ≤ q
        B[K] ← A[c]
        K ← K+1
        c ← c+1
    While J ≤ r
        B[K] ← A[J]
        K ← K+1
        J ← J+1
    ①
```

$C(n) = 2\,C(n/2) + C_{merge}(n)$

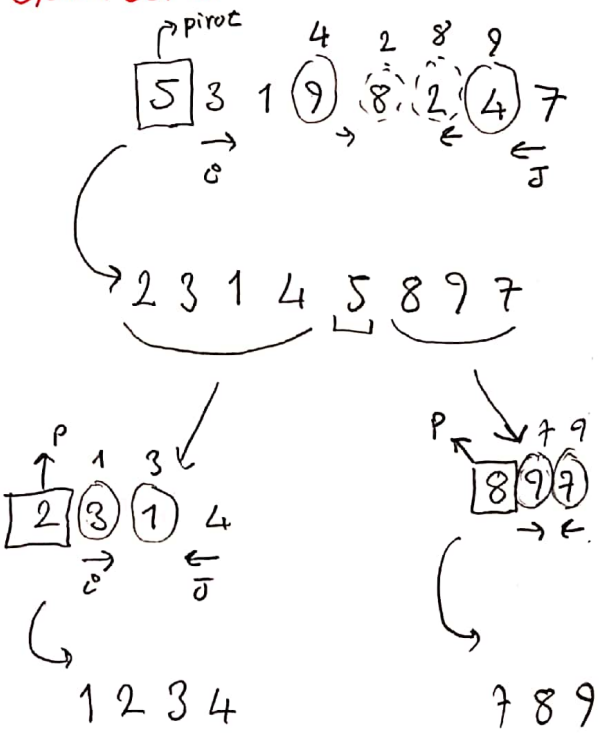
2 4 6 8
1 3 5 7 ⟶ compare 1, 2 · compare 2, 3 · compare 3, 4 · compare 4, 5 · compare 5, 6 · compare 6, 7 · compare 7, 8 ⟶ $n-1$

↳ worst

$C_{worst}(n) = 2C_{worst}(n/2) + n - 1$ ⟶ Worst case merge operation

$a=2, b=2, d=1 \longrightarrow \Theta(n^d \log n) \longrightarrow \Theta(n \log n)$

Quick Sort



5 9 8 7 6 4
5 4 8 7 6 9
5 4 6 7 8 9

```
QuickSort(A[l..r])
    if l < r
        s ← Partition(A[l..r])
        QuickSort(A[l..s-1])
        QuickSort(A[s+1..r])
```

Average Case

- Randomly ordered array of size n
- Assuming that partition split can happen in each position s $(0 \le s \le n-1)$ with the same probability $1/n$

$$C_{avg}(n) = \frac{1}{n} \cdot \sum_{s=0}^{n-1} [(n+1) + C_{avg}(s) + C_{avg}(n-1-s)] \quad \text{for } n > 1$$

$C_{avg}(0) = 0$

$C_{avg}(1) = 0$

$$C_{avg}(n) \approx 2n \ln n \approx 1.38\, n \log_2 n$$

↳ 38% more comparison

```
HoarePartition(A[l..r])
    p ← A[l]
    i ← l
    j ← r+1
    repeat
        repeat i ← i+1 until A[i] ≥ p
        repeat j ← j-1 until A[j] ≤ p
    until i ≥ j
    swap(A[i], A[j])
    swap(A[l], A[j])
    return j
```

$$C_{best}(n) = \underbrace{2 C_{best}(n/2)}_{f(n)} + n$$

↳ $\Theta(n \log n)$

comparison of n times

If the pivod is the median of array

↳ means best case

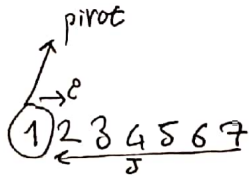
$$C_{worst}(n) = \underbrace{(n+1) + n + (n-1) + \cdots + 3}_{f(n)}$$

$$= \frac{(n+1)\cdot(n+2)}{2} - 3$$

$$\in \Theta(n^2)$$

last two element ↔ 2 comp.

1 2 → 1 compo



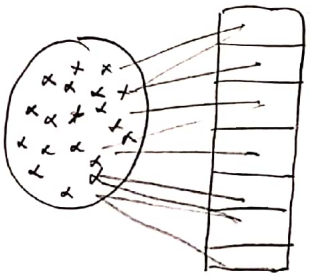
Left-to-right scan stops at $A[1]$

Right-to-left scan will go all the way to reach $A[0]$

Search Algorithms

- Linear (Sequential) Search → $\overbrace{O(n)}^{W/A/B}$
- Binary Search (sorted Dataset) → $\overbrace{O(\log_2 n)}^{W/A}$ $\overbrace{O(1)}^{B}$
- Binary Search Tree → $\overbrace{O(\log n)}^{B/A}$ $\overbrace{O(n)}^{W}$

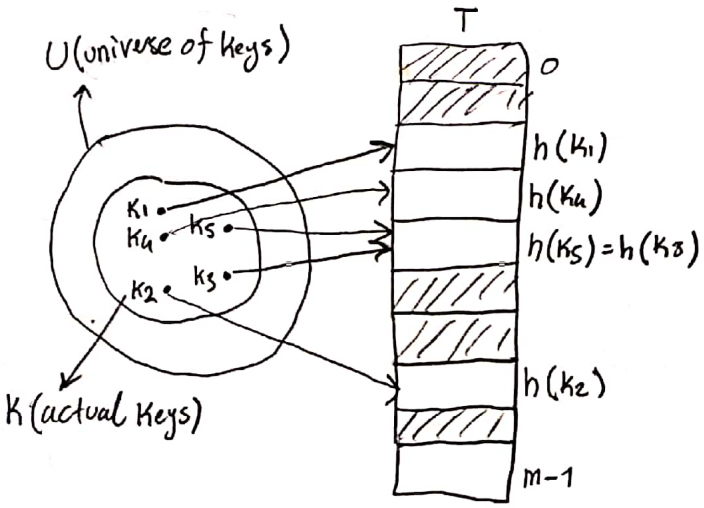
- Hashing
  - Direct Access Table



Hashing

Many applications require a dynamic set that supports only the dictionary operations; Insert, Search, Delete. A hash table is an effective data structure for implementing dictionaries.

Although searching for an element in a hash table can take as long as searching for an element in a linked list - $\Theta(n)$ time in worst case - in practice, hashing performs extremely well. Under reasonable assumptions, the average time to search for an element in a hash table is $O(1)$.



$h: U \longrightarrow \{0, 1, \ldots, m-1\}$

$|T| = |K| = m \rightarrow \Theta(|K|)$

↳ in direct addressing

$|T| = |U|$ → much more space

A good hash function satisfies (approximately) the assumption of simle uniform hashing: each key is equally likely to hash to any of the m slots, independently of where any other key hashed to.

The Division Method

We map a key k into one of m slots by taking the remainder of k divided by m.

$$h(k) = k \bmod m$$, where m is the size of hash table

When using the division method, we usually avoid certain values of m. For example, m should not be a power of 2, since $m = 2^p$, then h(k) is just the p lowest-order bits of k.

The Multiplication Method

Multiply key k by a constant A in the rauge of $0 < A < 1$ and extract fractional part of kA. Then multiply this value by m and take the floor of the result:

$$h(k) = \lfloor m * [(k * A) \bmod 1] \rfloor$$

$(k * A) \bmod 1$ means the fractional part of $k * A$:

$$(k * A) \bmod 1 = k * A - \lfloor k * A \rfloor$$

In multiplication method value of m is not crictial.
The optimal value of A depends on the characteristics of the data being hashed.

$A \approx (\sqrt{5}-1)/2 = 0.6180339887\ldots$

is likely to work reasonably well.

Birthday Paradox And Hashing

$$365 \rightarrow \frac{364}{365} * \frac{363}{365} * \cdots * \frac{365-(K-1)}{365}$$

↳ probability of K numbers not overlapping

Hash Table

Table size: m

$$\frac{m-1}{m} * \frac{m-2}{m} * \cdots * \frac{m-(K-1)}{m} \approx \exp\left(\frac{-K(K-1)}{m}\right)$$

} probability of K numbers not overlapping

$$\text{Hash collision Probability} = 1 - \exp\left(\frac{-K(K-1)}{m}\right)$$

Key

- Integer
- Floating Number → Convert to integer
- String

"ELMA", "MALE", "ALME" should not generate the same address.

a) $h \leftarrow 0$; for $c \leftarrow 0$ $S-1$ (↳ size of string) do

$h \leftarrow (h * C + ord(c)) \bmod m$ (↳ table length)

C ↳ Constant; ord(c) ↳ ASCII of char c

b) Horner's Method

$K = R^{L-1} * str[0] + \cdots + R^0 * str[L-1]$ (L ↳ length of string)

$R \rightarrow 31$; $str[0] \rightarrow str[0] - 'A' + 1$

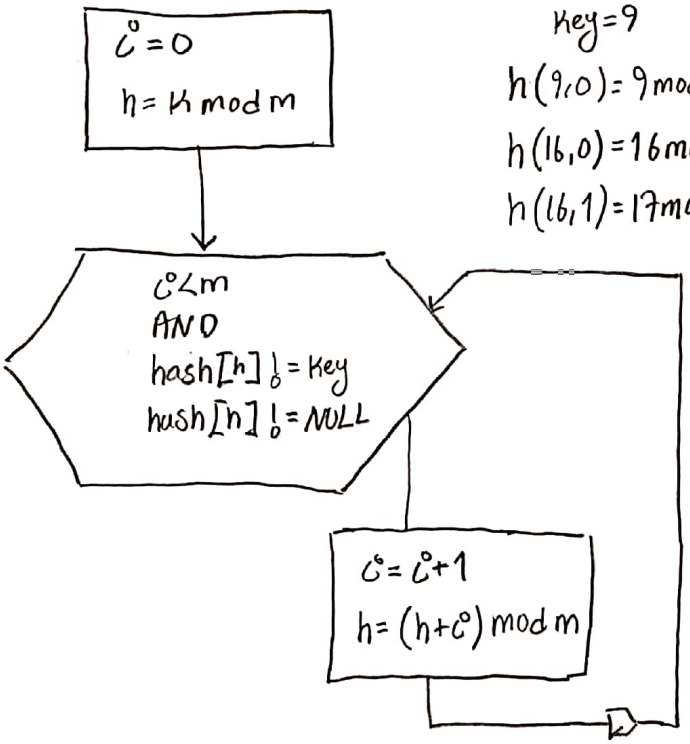
- Compound Keys

Birthdate: 31.08.1997 → $R * 31 + R^2 * 8 + R^3 * 1997$

Open Addressing

Linear Probing

$$h(K,i) = [h'(K) + i] \bmod m$$

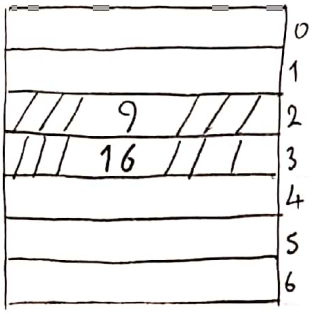


m=7
key=9

h(9,0) = 9 mod 7 = 2

h(16,0) = 16 mod 7 = 2

h(16,1) = 17 mod 7 = 3



- If the table is almost full, increase the table size (ReHashing).
- Delete operation → (-1) : Delete flag
- Linear probing suffers from a problem known as primary clustering. Long runs of occupied slots build up, increasing the average search time. Cluster arise because an empty slot preceded by i full slots gets filled next with probability (i+1)/m. Long runs of occupied slots tend to get longer, and the average search time increases.

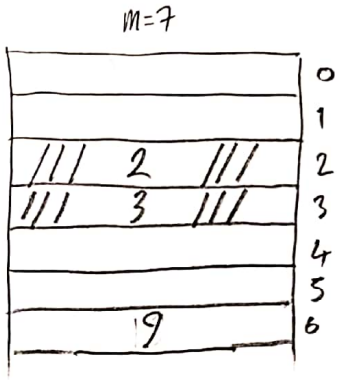
Quadratic Probing

- $h(K,i) = [h'(K) + i^2] \bmod m$

$h(9,0) = 9 \bmod 7 = 2$

$h(9,1) = 10 \bmod 7 = 3$

$h(9,2) = 13 \bmod 7 = 6$



- $h(K,i) = [h'(K) + c_1 i + c_2 i^2] \bmod m$

This method works much better than linear probing, but to make full use of hash table, the values of c_1, c_2 and m are constrained.

Also, if two keys have the same initial probe position, then their probe sequences are the same, since $h(K_1,0) = h(K_2,0)$, implies $h(K_1,i) = h(K_2,i)$. This property leads to a milder form of clustering, called secondary clustering.

Double Hashing

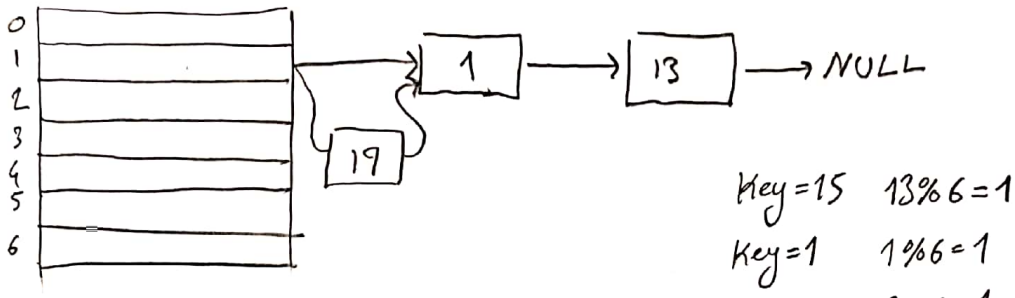
$$h(K,i) = [h_1(K) + i * h_2(K)] \bmod m$$

$$h_1(K) = K \bmod m$$

$$h_2(K) = 1 + (K \bmod m'), \quad m' < m$$

Double hashing offers one of the best methods available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Seperate Chaining



Key=15 13%6=1

Key=1 1%6=1

Key=19 19%6=1