# Asymptotic Analysis

`Limit Theorem of Order Notation:` Suppose for all $n \geq n_0$, we have $f(n), g(n) > 0$ and $L = \lim_n f(n)/g(n)$, then

$$f(n) \in \begin{cases} o(g(n)) & \text{if } L = 0 \\ \Theta(g(n)) & \text{if } 0 < L < \infty \\ \omega(g(n)) & \text{if } L = \infty \end{cases}$$

`Relations betwn Order Notations:`

- $f \in \Theta(g) \Leftrightarrow g \in \Theta(f)$;
- $f \in O(g) \Leftrightarrow g \in \Omega(f)$;
- $f \in o(g) \Leftrightarrow g \in \omega(f)$;
- $f \in o(g) \Rightarrow g \in O(f)$;
- $f \in \omega(g) \Rightarrow g \in \Omega(f)$;
- $f \in o(g) \Rightarrow g \notin \Omega(f)$;
- $f \in \omega(g) \Rightarrow g \notin O(f)$;

`Table of Recurrence Raltions:`

| Recursion | Resolves to |
|---|---|
| $T(n) \leq T(n/2) + O(1)$ | $T(n) \in O(\log n)$ |
| $T(n) \leq 2T(n/2) + O(n)$ | $T(n) \in O(n \log n)$ |
| $T(n) \leq 2T(n/2) + O(\log n)$ | $T(n) \in O(n)$ |
| $T(n) \leq cT(n-1) + O(1)$ for some $c < 1$ | $T(n) \in O(1)$ |
| $T(n) \leq 2T(n/4) + O(1)$ | $T(n) \in O(\sqrt{n})$ |
| $T(n) \leq T(\sqrt{n}) + O(\sqrt{n})$ | $T(n) \in O(\sqrt{n})$ |
| $T(n) \leq T(\sqrt{n}) + O(1)$ | $T(n) \in O(\log \log n)$ |

**We also have:**

$$(\log n)^c \in o(n^d)$$

for any $c$ and $d$ constants.

# Priority Queues

## Heapify and Heapsort

In heap, `insert()` and `deleteMax()` are both $O(\log n)$.

**Heapify:** Observe that if both subtrees of node $v$ have correct heap-order, fix-down on $v$ will establish correct order for the whole subtree of $v$, hence we have heapify pseudocode:

```
for i ← parent(last())$ down to 0: fix-down(A, i, n)
```

and the complexity of the given algorithm is $\Theta(n)$:

$$\sum_{i=0}^{h-1} 2^i(h-i) = 2^h \sum_{i=0}^{h-1} \frac{h-i}{2^{h-i}} = 2^h \sum_{i=1}^{h} \frac{i}{2^i}$$

**Heapsort:** Heapify and array, and keep swapping the root with the last element, hence sorting the array in non-decreasing order. Total time is $O(n \log n)$.

## Fine the $k^{th}$ smallest element

1. Make $k+1$ passes through the array, deleting the minimum number each time. Complexity: $\theta(kn)$.
2. Sort $A$, then return $A[k]$. Complexity: $n \log n$.
3. Create a min-heap with $heapify(A)$. Call $delete-min(A)$ $k+1$ times. Complexity: $n + k \log n$.

*$m$ is the number of digits

# Sorting, Average-Case and Randomized Algorithms

Define $T(I, R)$ to be running time of randomized algorithm for instance $I$ and $R$, sequence of random numbers algorithm choses. Then

**Expected Runtime for Instance $I$:**
$T_{exp}(I) = \mathbb{E}[T(I,R)] = \sum_R T(I,R) \cdot Pr(R)$; and

**Worst-case Expected Runtime:** $T_{exp}(n) = \max_{I \in I_n} T_{exp}(I)$.

We use randomized algorithms because we can improve running time and also improve solution. *It shift dependence from what we cannot control (user) to what we can control (RNG).*

Is expected time of randomized version always the same as average case time of non-randomized version?

- no in general (depends on randomization)
- yes if randomization is a shuffle, i.e., choose instance randomly with equal probability

## QuickSelect

QuickSelect has best case runtime $\Theta(n)$ and worst case runtime $\Theta(n^2)$, but the average case runtime is $\Theta(n)$. Hence we randomize QuickSelect so that the expected time of it is the same as the average case runtime.

## QuickSort

Best case runtime $\Theta(n \log n)$ and worst case runtime $O(n^2)$, but the average case runtime is $\Omega(n \log n)$.

## Comparison-Based Sorting

**Definition:** Sorting permutation stores array indexes in the order corresponding to the sorted array.

**Theorem:** Under comparison model, any sorting algorithm requires $\Omega(n \log n)$ comparisons.

## Non-Comparison-Based Sorting

These sortings are less general than comparison-based sortings.

## Bucket Sort

The runtime and auxiliary space for Bucket Sort are both $\Theta(n + R)$. Bucket sort is stable.

## MSD-Radix-Sort

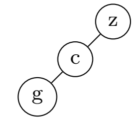
the total auxiliary space is $\Theta(n + R + m)$ *;
the total time is $O(mnR)$.

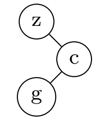
## LSD-Radix-Sort

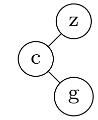
the total time is $P(m(n + R))$.

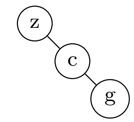
# Dictionaries

## Restructure AVL Tree

- 

u ← rotate-right(z)
- 

rotate-right(c);
u ← rotate-left(z)
- 

rotate-left(c);
u ← rotate-right(z)
- 

u ← rotate-left(z)

## AVL Tree Summary – Height is $\Theta(\log n)$

1. `Search` costs $\Theta(height)$;
2. `Insert` costs $\Theta(height)$, restructure will be called at most once;
3. `Delete` costs $\Theta(height)$, restructure may be called $\Theta(height)$ times.

# Other Dictionaries

**Skip List:** Expected length of list $S_i$ at height $i$ is $n/2^i$, expected height is $\leq 2 + \log n$. Expected space is $\Theta(n)$ and expected running time is $O(\log n)$.

**Biased Search:** Optimal static ordering – we know the access distribution; MTF (move-to-front) Heuristic; Transpose Heuristic.

# Dictionaries for Special Keys

Any comparison-based algorithm requires in the worst case $\Omega(\log n)$ comparisons to search among $n$ distinct items.

## Interpolation Search

$$\ell + \left\lfloor \frac{\overbrace{k - A[\ell]}^{\text{distance from left key}}}{\underbrace{A[r] - A[\ell]}_{\text{distance between left and right keys}}} \times \underbrace{(r - \ell - 1)}_{\text{\# unknown keys in range}} \right\rfloor$$

## Trie

The key function/ method for a trie is `Trie::get-path-to`, which returns a stack $P$ of all the ancestors of where $w$ would be stored.
In general, `search(w)`, `prefix-search(w)`, `insert(w)`, `delete(w)` all take time $\Theta(|w|)$.

**multiway trie** has bigger alphabet. Children can be stoed in

1. array;
2. linked list;
3. AVL tree

Arrays are fast, lists are space efficient, AVL tree is best in theory, but not worth it in practice unless $|\Sigma|$ is huge.

**Load Factor**, $\alpha$, is defined to be $\frac{n}{M}$, where $n$ is the number of elements and $M$ is the size of the hash table.
**Linear Probing**: Expected runtime of `search` and `delete` is $\Theta(1+\alpha)$, `insert` is $\Theta(1)$, space is $O(M+n)$.
For cuckoo hashing, the **load factor** is defined as $\frac{n}{|T_0|+|T_1|}$, and if it is small enough, $\alpha < 1/2$, then insertion has $O(1)$ expected time (but this wastes space, expected space is $O(n)$).

1. All strategies have $O(1)$ expected time for `search`, `insert`, `delete`
2. Cuckoo hashing has $O(1)$ worst case for `search`, `delete`
3. Probe sequence use $O(n)$ worst case space
4. Cuckoo hashing uses $O(n)$ expected space

For any hashing, the worst case runtime for `insert` is $\Theta(n)$.

## Range Search

**Quad Tree**: Height of quad tree is proven $\Theta(\rho(S))$, where $\rho(S)$ is the **spread factor** defined to be $L/d_{min}$.
The complexity to build the initial tree and perform range search are both $\Theta(nh)$.

**kd-Tree**: Consider a kd-Tree for $d$-dimensional space, we have

$$\begin{aligned} \texttt{storage} &: O(n); \\ \texttt{height} &: O(\log n); \\ \texttt{Construction Time} &: O(n \log n); \\ \texttt{Range query time} &: O\left(s + n^{1-1/d}\right), \quad d \text{ is a constant.} \end{aligned}$$

**Range Tree**: Range trees can be generalized to d -dimensional space:

$$\begin{aligned} \texttt{storage} &: O(n(\log n)^{d-1}); \\ \texttt{Construction Time} &: O(n(\log n)^{d}); \\ \texttt{Range query time} &: O\left(s + (\log n)^{d}\right) \end{aligned}$$

## String Matching

**Karp Rabin**: use hash values (called fingerprints) to eliminate guesses.
Given that we can compute the next hash value from the previous one, we can show that expected running time is $O(m+n)$. Although $\Theta(mn)$ is the worst-case, but this is extremely unlikely.
**KMP**: *Failure array*, store the length of the longest valid suffix of $P[1,\ldots,j]$ in $F[j]$.

```
F←failure-array(P), i←0, j←0
while i < n do
    if P[j] = T[i]
        if j = m-1: "found at i - j"
        else: i←i+1, j←j+1
    else // $P[j] \neq T[i]$
        if j > 0: j←F[j-1]
        else: i←i+1
return FAIL
```

Failure array $O(m)$, matching $O(n)$, so $O(m+n)$ in total.
**Boyer-Moore**: Reverse order searching, bad character heuristic, last Occurrence Array,

```
L←last occ arr of P, j←m-1, i←m-1
while i<n and j≥0 do // curr guess @ i-j
    if T[i] = P[j] then: i←i-1, j←j-1
    else: i←i + m-1 - min{L(c), j-1}
            j←m-1
if j = -1 return "found at guess i + 1" else return FAIL
```

**Suffix Tree**: build suffix tree by inserting each suffixes of $T$ into a compressed trie ($\Theta(|\Sigma|n^2)$ but there is a $\Theta(|\Sigma|n)$ way).
`prefix-search` for $P$ in the trie is $O(|\sigma|m)$ if children are stored in linked list, or $O(m)$ if in array.
**Suffix Array**: storing sorted permutation of the suffixes of $T$ (implicitly, by storing start indices).

## Compression

**Huffman Tree**: assign weight to each trie based on the frequency and merge the two with the smallest frequencies until only one left. Total encoding time is $O(|\Sigma_S|\log|\Sigma_S| + |C|)$. Decoding run-time: $O(|C|)$.
**LZW**: algorithm discovers and encodes frequent substring as we process text, no need to know frequent substrings beforehand. When decoding, rememebr

$$s = s_{prev} + s_{prev}[0]$$

**bzip2**:

`BWT` → `MTF` → `0-run encoding` → `Huffman`

| Huffman | Lempel-Ziv-Welch | bzip2 (uses Burrows-Wheeler) |
|---|---|---|
| variable-length | fixed-width | multi-step |
| single-character | multi-character | multi-step |
| 2-pass, must send dictionary | 1-pass | not streamable |
| requires uneven frequencies | requires repeated substrings | requires repeated substrings |

**2-4 Tree**: each node contains one, two, or three KVPs. All empty subtrees are at the same level. **`delete:`**

```
24Tree::delete(k)
    v ← 24Tree::search(k)
    if v is not a leaf
        swap k with its inorder successor k'
        swap v with leaf that contained k'
    delete k and one empty subtree in key-subtree-list of v
    while v has 0 keys  // underflow
        if v is the root, delete v and break
        if v has immediate sibling u with 2 or more KVPs  // transfer,
              then done!
            transfer the key of u that is nearest to v to p
            transfer the key of p between u and v to v
            transfer the subtree of u that is nearest to v to v
            break
        else  // merge and repeat
            u ← immediate sibling of v
            transfer the key of p between u and v to u
            transfer the subtree of v to u
            delete node v
            v ← p
```

## External Memory

**(a,b)-tree**: we have $b \geq 3$, $2 \leq a \leq \lceil b/2 \rceil$,

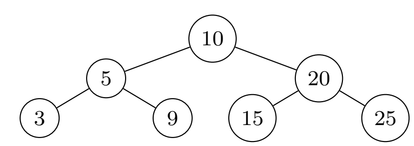
$$a \leq \texttt{\# subtree} \leq b$$

but root can have $\geq 2$ subtrees. All empty nodes are at the same level. $h \in O(\log_a n)$, right if $a \approx b/2$.

# of KVPs is equal to # of $\varnothing$'s.

**B-tree**: Special kind of of (a,b)-tree with $a = \lceil b/2 \rceil$, and so $h \in \Theta(\log_B n)$ (transfers).

**Tree Traversals**:




| | |
|---|---|
| `LEVEL-ORDER` | $[[10],[5,20],[3,9,15,25]]$ |
| `PRE-ORDER` | $[10,5,3,9,20,15,25]$ |
| `IN-ORDER` | $[3,5,9,10,15,20,25]$ |
| `POST-ORDER` | $[3,9,5,15,25,20,10]$ |