Array

Referential array. Array of object references.

Figure 5.4: An array storing references to strings.

E.g., Python lists, tuples.

Compact array. Array that store bits representing primary data.

Dynamic array. Resizable array that grows or shrinks based on the number of items it contains, so that its operations can have amortized $O(1)$ runtime.

E.g., Python list is implemented using dynamic array.

Operation	Runtime
`data[j] = val`	$O(1)$
`data.append(value)`	$O(1)$*
`data.insert(k, value)`	$O(n-k+1)$* (element shifts)
`data.pop()`	$O(1)$*
`data.pop(k)` `del data[k]`	$O(n-k)$* (element shifts)
`data.remove(value)`	$O(n)$*
`data1.extend(data2)` `data1 += data2`	$O(n_2)$*
`data.reverse()`	$O(n)$
`data.sort()`	$O(n \log n)$

- Amortized.

Reversing list:

```
# let rev list =
    let rec aux acc = function
      | [] -> acc
      | h::t -> aux (h::acc) t in
    aux [] list;;
```

Stack

Method	Description	Runtime
`S.push(e)`	Add e to top of stack.	$O(1)$*
`S.pop()`	Remove and return item from top of stack.	$O(1)$*
`S.top()`	Return reference to item at top of stack.	$O(1)$
`S.is_empty()`	True if the stack is empty.	$O(1)$
`len(S)`	Return the number of items in the stack.	$O(1)$

*If implemented using a Python list, these operations are amortized.

Applications:

1. Reverse a list (push all items in and pop them one by one, first in last out).
2. Parenthesis matching.

Queue

Method	Description	Runtime
`Q.enqueue(e)`	Add `e` to end of queue.	$O(1)$*
`Q.dequeue()`	Remove and return item from front of queue.	$O(1)$*
`Q.first()`	Return item at front of queue.	$O(1)$
`Q.is_empty()`	True if the queue is empty.	$O(1)$
`len(Q)`	Return the number of items in the queue.	$O(1)$

*If implemented using a Python list (circular, wraps around when reaching end of list), these operations are amortized.

Deque

Double-ended queue. ADT that can add and remove elements from both ends of the queue.

Method	Description	Runtime
`D.add_first(e)`, `D.add_last(e)`	Add `e` to front/back of dequeue.	$O(1)$*
`D.delete_first(e)`, `D.delete_last(e)`	Remove and return item from front/back of dequeue.	$O(1)$*
`D.first()`, `D.last()`	Return and return item at the front/back of dequeue.	$O(1)$
`D.is_empty()`	True if the dequeue is empty.	$O(1)$
`len(D)`	Return the number of items in the dequeue.	$O(1)$

*If implemented using a Python list (circular, wraps around), these operations are amortized.

Linked List

Singly Linked List

Applications:

1. Implement the Stack ADT, all operations are worst-case $O(1)$.
2. Implement the Queue ADT, all operations are worst-case $O(1)$.

Circularly Linked List

Figure 7.7: Example of a singly linked list with circular structure.

Applications:

1. Implement the Queue ADT, with more efficient method for wrapping around.

Doubly Linked List

Figure 7.10: A doubly linked list representing the sequence { JFK, PVD, SFO }, using sentinels header and trailer to demarcate the ends of the list.

Applications:

1. Implement the Deque ADT.
2. Implement the Positional List ADT.

Positional List

Figure 7.14: We wish to be able to identify the position of an element in a sequence without the use of an integer index.

Method	Description
`L.first()`, `L.last()`	Return the position of the first/last item.
`L.before(p)`, `L.after(p)`	Return the position immediately before/after position `p`.
`L.is_empty()`	True if the positional list is empty.
`len(L)`	Return the number of items in the positional list.
`iter(L)`	Return a forward iterator of items in the positional list.
`L.add_first(e)`, `L.add_last(e)`	Add `e` to the front/back of the positional list.
`L.add_before(p, e)`, `L.add_after(p, e)`	Add `e` before/after position `p`.
`L.replace(p, e)`	Replace the item at position `p` with `e`.
`L.delete(p)`	Remove and return the item at position `p`.

Applications:

1. Maintain access frequencies.

Array-based vs. Linked-based

Metrics	Array-based	Link-based
access based on index	$O(1)$	$O(n)$
search	$O(\log n)$ if sorted (binary search)	$O(n)$
insertion, deletion	$O(n)$ worst case (need to shift elements)	$O(1)$ at arbitrary position
memory usage	$2n$ worst case (after resize)	$2n$ for singly-linked lists $3n$ for doubly-linked lists

Compromise between array-based and link-based structures: Skip lists achieve average $O(\log n)$ search and update operations via a probabilistic method.

Tree

General Tree

Tree. A tree T is set of nodes storing elements such that the nodes have a parent-child relationship that satisfies the following properties:

- If T is nonempty, it has a special node, called the root of T , that has no parent.
- Each node v of T different from the root has a unique parent node w; every node with parent w is a child of w.

Sibling. Two nodes are siblings if they have the same parent node.

External. A node is external if it has no children. A.k.a leaves.

Internal. A node is internal if it has >= 1 children.

Edge. An edge of tree T is a pair of nodes (u, v) such that u is the parent of v, or vice versa.

Path. A path of T is a sequence of nodes such that any two consecutive nodes in the sequence form an edge.

Ordered Tree. A tree is ordered if there is a meaningful linear order among the children of each node.

Depth of node. The depth of a node is the number of its ancestors, excluding itself.

Depth of node (recursive). If p is the root, then its depth is 0. Otherwise, the depth of p is $1+$ depth of p's parent.

Height of node (recursive). If p is a leaf, then its height is 0. Otherwise, the height of p is $1+$ the maximum of p's children's heights.

Height of tree. The height of a tree is the height of its root.

Proposition. The height of a nonempty tree is the maximum of its leaves' depths.

Proposition. In a tree with n nodes, the sum of the number of children of all nodes is $n - 1$.

Proof. Every node except for the root is some other node's child.

Method	Description	Runtime
`T.root()`	Return the position of the tree's root.	
`T.is_root(p)`	True if position `p` is the tree's root.	
`T.parent(p)`	Return the position of `p`'s parent.	
`T.num children(p)`	Return the number of `p`'s children.	
`T.children(p)`	Generate an iteration of position `p`'s children.	
`T.is leaf(p)`	True if position `p` does not have any children.	
`len(T)`	Return the number of positions in the tree.	
`T.is empty()`	True if the tree does not contain any position.	
`T.positions()`	Generate an iteration of the positions in the tree.	
`iter(T)`	Generate an iteration of the elements in the tree.	
`T.depth(p)`	Return the depth of `p`.	$O(d_p + 1)$
`T.height()`	Return the height of `T`.	$O(n)$*

*`T.height()` computes height recursively, and it takes $O(\sum_p (1 + c_p)) = O(n + \sum_p c_p)$ (c_p is number of `p`
s children), which is $O(n + (n - 1)) = O(n)$ time by the above proposition.

Binary Tree

Binary tree. A binary tree is an ordered tree such that:

1. Every node has at most two children.
2. Each child node is either a left child or a right child.
3. A left child precedes a right child in the order of children of a node.

Binary tree (recursive). A binary tree is either empty or consists of:

- A node r, called the root of T, that stores an element.
- A binary tree (possibly empty), called the left subtree of T.
- A binary tree (possibly empty), called the right subtree of T.

Method	Description
`T.left(p)`, `T.right(p)`	Return the position of `p`'s left/right child.
`T.sibling(p)`	Return the position of `p`'s sibling.

Proper/Full. A binary tree is proper or full if each node has either zero or two children. That is, all its internal nodes have two children.

Proposition. In a nonempty proper binary tree T, with n_E external nodes and n_I internal nodes, we have $n_E = n_I + 1$.

Proof. If h is T's height, then $n_E = 2^h$, $n_I = 2^0 + 2^1 + 2^2 + \cdots 2^h = 2^h - 1$.

Implementations

Linked Binary Tree

Figure 8.11: A linked structure for representing: (a) a single node; (b) a binary tree.

Method	Description
`T.add_root(e)`	Add root `e` to an empty tree.
`T.add_left(p, e)`, `T.add_right(p, e)`	Add `e` as left/right child to `p`.
`T.replace(p, e)`	Replace element at position `p` with `e`.
`T.delete(p)`	Remove the node at position `p` and replace it with its only child.
`T.attach(p, T1, T2)`	Attach `T1`, `T2` as left and right subtress of the leaf `p`.

Operation	Runtime
`len`, `is_empty`	$O(1)$
`root`, `parent`, `left`, `right`, `sibling`, `children`, `num_children`	$O(1)$
`is_root`, `is_leaf`	$O(1)$
`depth(p)`	$O(d_p + 1)$
`height`	$O(n)$
`add_root`, `add_left`, `add_right`, `replace`, `delete`, `attach`	$O(1)$

Array-based Binary Tree

For every position p of T , let $f(p)$ be the integer defined as follows.

- If p is the root of T, then $f(p) = 0$.
- If p is the left child of position q, then $f(p) = 2f(q) + 1$.
- If p is the right child of position q, then $f(p) = 2f(q) + 2$.

An array-based structure A (such as a Python list) of capacity N, with the element at position p of T stored at $A[f(p)]$.

Figure 8.13: Representation of a binary tree by means of an array.

Drawbacks:

- Space usage depends on the binary tree's shape.
  - Worst case: All nodes in the right subtree of previous node, $N = 2^n - 1$.
  - Best case: Binary tree is complete (e.g., heap), $N = n$.
- `delete` is $O(n)$ as all the node's descendants need to be shifted in the array.

Linked General Tree

Figure 8.14: The linked structure for a general tree: (a) the structure of a node; (b) a larger portion of the data structure associated with a node and its children.

Operation	Runtime
`len`, `is_empty`	$O(1)$
`root`, `parent`, `is_root`, `is_leaf`	$O(1)$
`children(p)`	$O(c_p + 1)$
`depth(p)`	$O(d_p + 1)$
`height`	$O(n)$

Tree Traversal Algorithms

Traversals are $O(n)$ as they must visit every node in the tree.

Binary search is $O(\log n)$ in a proper binary tree.

Preorder (General Tree)

Visit node, then visit node's children.

```
Algorithm preorder(T, p):
  perform the "visit" action for position p
  for each child c in T.children(p) do
    preorder(T, c)                    {recursively traverse the subtree rooted at c}
```

Code Fragment 8.12: Algorithm preorder for performing the preorder traversal of a subtree rooted at position p of a tree T.

Figure 8.15 portrays the order in which positions of a sample tree are visited during an application of the preorder traversal algorithm.

Figure 8.15: Preorder traversal of an ordered tree, where the children of each position are ordered from left to right.

Postorder (General Tree)

Visit node's children, then visit node.

```
Algorithm postorder(T, p):
  for each child c in T.children(p) do
    postorder(T, c)                   {recursively traverse the subtree rooted at c}
  perform the "visit" action for position p
```

Code Fragment 8.13: Algorithm postorder for performing the postorder traversal of a subtree rooted at position p of a tree T.

Figure 8.16: Postorder traversal of the ordered tree of Figure 8.15.

Breadth-first (General Tree)

Visit nodes level by level.

Not recursive.

Figure 8.17: Partial game tree for Tic-Tac-Toe, with annotations displaying the order in which positions are visited in a breadth-first traversal.

Dequeue to get node. Visit node, then enqueue node's children.

```
Algorithm breadthfirst(T):
  Initialize queue Q to contain T.root()
  while Q not empty do
    p = Q.dequeue()                    {p is the oldest entry in the queue}
    perform the "visit" action for position p
    for each child c in T.children(p) do
      Q.enqueue(c)     {add p's children to the end of the queue for later visits}
```

Code Fragment 8.14: Algorithm for performing a breadth-first traversal of a tree.

Inorder (Binary Tree)

Visit left subtree. Visit right subtree. Visit node.

```
Algorithm inorder(p):
  if p has a left child lc then
    inorder(lc)                    {recursively traverse the left subtree of p}
  perform the "visit" action for position p
  if p has a right child rc then
    inorder(rc)                    {recursively traverse the right subtree of p}
```

Code Fragment 8.15: Algorithm inorder for performing an inorder traversal of a subtree rooted at position p of a binary tree.

Figure 8.18: Inorder traversal of a binary tree.

Priority Queue

Method	Description
`P.add(k, v)`	Add item with key `k` and value `v` into the priority queue.
`P.min()`	Return item with the minimum key in the priority queue.
`P.remove_min()`	Remove and return an item with the minimum key in the priority queue.
`P.is_empty()`	True if the priority queue is empty.
`len(P)`	Return the number of items in the priority queue.

Unsorted Priority Queue

Add item to the end of the priority queue, find minimum to remove in $O(n)$.

$O(1)$ insertions, $O(n)$ removals (best-case, because it always takes $O(n)$ to find the minimum).

Method	Runtime
`P.add(k, v)`	$O(1)$
`P.min()`	$O(n)$
`P.remove_min()`	$O(n)$
`P.is_empty()`	$O(1)$
`len(P)`	$O(1)$

Sorted Priority Queue

Maintain sortedness when inserting items, minimum is at front of the priority queue,

$O(n)$ insertions (best-case is $O(1)$, because the items may come in as sorted), $O(1)$ removals.

Method	Runtime
`P.add(k, v)`	$O(n)$
`P.min()`	$O(1)$
`P.remove_min()`	$O(1)$
`P.is_empty()`	$O(1)$
`len(P)`	$O(1)$

Sorting with Priority Queue

1. Add each item one by one to the priority queue.
2. Keep removing the minimum from the priority queue.

Implementation/Operation	`add`	`remove_min`
Unsorted List	Add to the end of list in $O(1)$.	Find minimum to remove in best-case $O(n)$.
Sorted List	Insert into sorted list in $O(n)$ (best-case $O(1)$).	Remove minimum from front of list in $O(1)$.

With unsorted list-based priority queue, this is selection-sort (best case $O(n^2)$). With sorted list-based priority queue, this is insertion-sort (best-case $O(n)$).

Adaptable Priority Queue

Additional operations:

1. Remove arbitrary entry.
2. Update the key (priority) of exiting entry.

Heap

A heap is a binary tree T that satisfies the following:

Heap-Order Property. (relational): In a heap T, for every position p other than the root, the key stored at p >= the key stored at p's parent. (So the minimum is at the heap's root.)

Complete Binary Tree Property. (structural): A heap T with height h is a complete binary tree if levels $0, 1, 2, \ldots, h-1$ of T have the maximum number of nodes possible and the remaining nodes at level h reside in the leftmost positions.

Proposition. A heap T storing n entries has height $h = \text{floor}(\log n)$.

Heap-based Priority Queue

Add:

1. Add item to rightmost node at bottom level to maintain completeness.
2. Up-heap bubbling to swap the new node into correct position to maintain heap-order.

Remove:

1. Remove minimum item from top of heap (root).
2. Copy item at rightmost node at bottom level to root.
3. Down-heap bubbling to swap the node into correct position to maintain heap order.

Method	Runtime
`P.add(k, v)`	$O(\log n)$*
`P.min()`	$O(1)$
`P.remove_min()`	$O(\log n)$*
`P.is_empty()`	$O(1)$
`len(P)`	$O(1)$

*For array-based-tree-based heap, this is amortized, but array-based heap can locate last position in $O(1)$ via index access.

Implementation/Operation	`add`	`remove_min`
Unsorted List	Add to the end of list in $O(1)$.	Insert into sorted list in $O(n)$.
Sorted List	Find minimum in $O(n)$ to remove.	Remove minimum from front of list in $O(1)$.
Array-based Heap	1. Find last position in $O(1)$. 2. Up-heap bubbling in $O(\log n)$.	1. Remove minimum at root in $O(1)$. 2. Find last position and copy to root in $O(1)$. 3. Down-heap bubbling in $O(\log n)$.
Linked Heap	1. Find last position in $O(\log n)$. 2. Up-heap bubbling in $O(\log n)$.	1. Remove minimum at root in $O(1)$. 2. Find last position and copy to root in $O(\log n)$. 3. Down-heap bubbling in $O(\log n)$.

Heap-Sort

Since `add` and `remove_min` are both $O(\log n)$ for heap-based priority queue, heap-sort is $O(n \log n)$.

In-place heap-sort (can sort in-place because heap is complete binary tree which doesn't have gaps in array-based representation):

Example: Note | indicates the division between heap and in/outS

```
input: |4  7  2  1  3
Remove from sequence and place in heap:
heap | inS
4|7  2  1  3
7  4|2  1  3
7  4  2|1  3
7  4  2  1|3
7  4  2  1  3|

Remove min from heap and place in output sequence:
heap | outS
4  2  3  1|7
3  2  1|4  7
2  1|3  4  7
1|2  3  4  7
Output:  1  2  3  4  7|
```

Figure 9.9: Phase 2 of an in-place heap-sort. The heap portion of each sequence representation is highlighted. The binary tree that each sequence (implicitly) represents is diagrammed with the most recent path of down-heap bubbling highlighted.

Map

Python uses dictionary to represent namespace. Can assume $O(1)$ lookup time.

Hash Table

Hash table is a lookup table structure that supports $O(1)$ lookup when implementing maps. The fast lookup is made possible by directly computing the hash table index from the map's key k via a hash function $h(k)$.

Hash Functions

We store the item (k, v) in the bucket $A[h(k)]$.

A hash function has two parts:

1. Hash code: keys $\to \mathbb{Z}$.
2. Compression function: $\mathbb{Z} \to [0, N-1]$ (N is the number of entries in the hash table).

Figure 10.5: Two parts of a hash function: a hash code and a compression function.

Hash codes

1. Interpret keys as integers. If overflow, sum the upper and lower 32-bits, or take bitwise exclusive-or. Does not preserve meaningful order in the key's individual components (e.g., characters in strings), if any.
2. Polynomial in nonzero constant a. $x_0 a^{n-1} + x_1 a^{n-2} + \cdots + x_{n-2} a + x_{n-1}$.
3. Cyclic shift. Sum each character in the key and shift the partial sum's bits cyclically in between.

Compression functions

1. "Division Method": $x \to x \mod N$.
2. "MAD Method": $x \to [(ax + b) \mod p] \mod N$.

Collision-handling schemes

Separate Chaining

Open Addressing

Linear Probing

Iteratively tries the buckets $A[(h(k)+i) \mod N]$, for $i = 0, 1, 2, \ldots$, until finding an empty bucket.

Figure 10.7: Insertion into a hash table with integer keys using linear probing. The hash function is $h(k) = k \bmod 11$. Values associated with keys are not shown.

Quadratic Probing

Iteratively tries the buckets $A[(h(k)+f(i)) \mod N]$, for $i = 0, 1, 2, \ldots$, where $f(i) = i^2$, until finding
an empty bucket.

Double Hashing

For secondary hash function $h'(i)$, iteratively tries the buckets $A[(h(k)+f(i)) \mod N]$, for $i = 0, 1, 2, \ldots$, where $f(i) = i \cdot h'(i)$, until finding an empty bucket.

Rehashing

Load factor. If n is the number of entries in a bucket array of capacity N, then the hash table's load factor is $\lambda = n/N$.

For each collision-handling scheme, there's a load factor threshold. If the load factor exceeds the threshold, the lookup efficiency starts degrading. For separate chaining, this is 0.9, linear probing 0.5, and Python dictionary's opening addressing $2/3$.

After the threshold is exceeded, the hash table is usually resized to twice the capacity (and all entries rehashed) to restore efficiency.

Sorted Search Table

A hash table where keys are sorted. The sortedness of the keys support inexact search such as searching for a range of keys.

Array-based implementation of the sorted search table allows $O(\log n)$ search via binary search, though update operations are $O(n)$ because elements need to be shifted.

Set

Set. An unordered collection of elements, without duplicates, that typically supports efficient membership tests (e.g., using hash tables).

Sets are implemented using hash tables in Python. In fact, they are like maps with keys without values.

Multiset. A multiset is like a set but allows duplicates.

Multimap. A multimap is like a map but allows the same key to map to multiple values.

Operation	Description
`S.add(e)`	Add `e` to the set if it's not yet in the set.
`S.discard(e)`	Remove `e` from the set if it's in the set.
`e in S`	True if `e` is in the set.
`len(S)`	Return the number of elements in the set.
`iter(S)`	Return an iteration of the elements in the set.

Search Tree

Binary Search Tree

Binary search tree. A binary tree T with each position p storing a key-value pair (k, v) such that:
• Keys in p's left subtree are $< k$.
• Keys in p's right subtree are $> k$.

Proposition. An inorder traversal of a binary search tree visits positions in increasing order of their keys.

Successor of node. The successor of the node at position p is the node with the smallest value that is $>= p$ (the next node to visit right after p in an inorder traversal).

- If p has a right subtree, this is the leftmost node in its right subtree.
- Else, this is the nearest ancestor such that p is in its left subtree.

Predecessor of node. The predecessor of the node at position p is the node with the largest value that is $<= p$ (the node visited right before p in an inorder traversal).

- If p has a left subtree, this is the rightmost node in its left subtree.
- Else, this is the nearest ancestor such that p is in its right subtree.

Deletion

If node to delete has only one child, replace it with its child:

Figure 11.5: Deletion from the binary search tree of Figure 11.4b, where the item to delete (with key 32) is stored at a position p with one child r: (a) before the deletion; (b) after the deletion.

If node to delete has two children,

1. replace it with its predecessor, and
2. replace its predecessor with its predecessor's child (the case above):

Figure 11.6: Deletion from the binary search tree of Figure 11.5b, where the item to delete (with key 88) is stored at a position p with two children, and replaced by its predecessor r: (a) before the deletion; (b) after the deletion.

Efficiency

Efficiency of binary search in a binary search tree depends on its height.

Figure 11.3: Illustrating the running time of searching in a binary search tree. The figure uses standard caricature of a binary search tree as a big triangle and a path from the root as a zig-zag line.

This motivates the following methods to balance a binary search tree.

Balanced Search Tree

Rotation

Rotating y above parent z, with x remaining as y's subtree.

Rotating x above parent y, then above grandparent z (with y remaining as x's subtree).

After the first rotation, the first tree becomes

```
a=z
    b=x
        c=y
```

And the second tree becomes

```
        c=z
    b=x
a=y
```

Rotating x above z gives the result (the relation between x and y remains unchanged in this rotation).

AVL Tree

An AVL tree is a binary search that satisfies the following:

Height balance property. For every position p of T, the heights* of the children of p differ by at most 1.

***Height of node (alternate definition).** Number of nodes (instead of edges) in a longest path from p to a leaf. E.g., height of a leaf is 1.

Figure 11.11: An example of an AVL tree. The keys of the items are shown inside the nodes, and the heights of the nodes are shown above the nodes (with empty subtrees having height 0).

Proposition. The height of an AVL tree storing n entries is $O(\log n)$.

Balanced node. In a binary search tree T, a position is balanced if the absolute value of the difference between the heights of its children is at most 1.

So the height balance property <=> every node is balanced.

Figure 11.12: An example insertion of an item with key 54 in the AVL tree of Figure 11.11: (a) after adding a new node for key 54, the nodes storing keys 78 and 44 become unbalanced; (b) a trinode restructuring restores the height-balance property. We show the heights of nodes above them, and we identify the nodes x, y, and z and subtrees T_1, T_2, T_3, and T_4 participating in the trinode restructuring.

Figure 11.14: Deletion of the item with key 32 from the AVL tree of Figure 11.12b: (a) after removing the node storing key 32, the root becomes unbalanced; (b) a (single) rotation restores the height-balance property.

For deletion, need to continue walking upward to repair further unbalances, until reaching the root. The number of operations is bounded by tree height O(log n).

So AVL tree guarantees $O(\log n)$ bound for binary search tree operations.

Figure 11.15: Illustrating the running time of searches and updates in an AVL tree. The time performance is $O(1)$ per level, broken into a down phase, which typically involves searching, and an up phase, which typically involves updating height values and performing local trinode restructurings (rotations).

Splay Tree

Moves more frequently accessed nodes closer to the root.

Amortized $O(\log n)$ for search, insertions, deletions.

(2, 4) Tree

Particular case of a multiway search tree, where each node may have multiple children. Each internal node has 2, 3, or 4 children.

Operations' runtime is the same as AVL tree.

Red-Black Tree

Only requires $O(1)$ structural operations for rebalancing (instead of $O(\log n)$ as in AVL tree).

A red-black tree is a binary search tree with nodes colored red and black such that:

Root Property. The root is black.

Red Property. The children of a red node (if any) are black.

Depth Property. All nodes with zero or one children have the same black depth, defined as the number of black ancestors. (Recall that a node is its own ancestor).

Proposition. The height of a red-black tree storing n entries is $O(\log n)$.

Sorting

Runtime of sorting algorithms is bounded from above by $O(n!)$, because sorting produces a permutation of the original sequence, and there are $n!$ permutations.

Runtime of comparison-based sorting is bounded from below by $\Omega(n \log n)$.

$$\log(n!) \geq \log\left(\left(\frac{n}{2}\right)^{\frac{n}{2}}\right) = \frac{n}{2}\log\frac{n}{2},$$

which is $\Omega(n \log n)$. ■

Figure 12.15: Visualizing the lower bound for comparison-based sorting.

Insertion-Sort

Maintain a sorted sub-list at the front, keep inserting the next element from the unsorted sub-list into the sorted sub-list to make sure it stays sorted.

Scenario	Runtime	Description
Worst case	$O(n^2)$	When input is sorted in reverse.
Best case	$O(n)$	When input is sorted.
Average	$O(n^2)$	

Algorithm InsertionSort(A):
  Input: An array A of n comparable elements
  Output: The array A with elements rearranged in nondecreasing order
  for k from 1 to n − 1 **do**
    Insert A[k] at its proper location within A[0], A[1], …, A[k].

Code Fragment 5.9: High-level description of the insertion-sort algorithm.

Selection-Sort

Maintain a sorted sub-list at the front, keep selecting the minimum from the unsorted sub-list and swapping it to the end of the sorted sub-list.

Scenario	Runtime	Description
Worst case	$O(n^2)$	
Best case	$O(n^2)$	Even when input is sorted.
Average	$O(n^2)$	

Bubble-Sort

Make n passes, swapping every two items into order in each pass.

Scenario	Runtime	Description
Worst case	$O(n^2)$	When input is sorted in reverse.
Best case	$O(n)$	When input is sorted.
Average	$O(n^2)$	

Heap-Sort

First, add each item to heap. Then, keep popping the minimum from the heap.

With unsorted/sorted list-based heaps, this is selection/insertion-sort.

With linked/array-based binary tree-based heaps, this is heap-sort.

Scenario	Runtime	Description
Worst case	$O(n \log n)$	
Best case	$O(n \log n)$	If input is sorted, adding items to heap is $O(1)$ with array-based heaps, but down-heap bubbling when removing items is still $O(\log n)$.
Average	$O(n \log n)$	

Merge-Sort

Divide and conquer.

1. Divide into two halves.
2. Recursively sort on each half.
3. Merge the sorted halves into a sorted list (where hard work is done).

Scenario	Runtime	Description
Worst case	$O(n \log n)$	
Best case	$O(n \log n)$	
Average	$O(n \log n)$	

There are $\log n$ levels, and the merge step at each level is $O(n)$ ($O(n/2 * 2)$, $O(n/4 * 4)$, etc). So merge-sort is $O(n \log n)$.

Figure 12.6: A visual analysis of the running time of merge-sort. Each node represents the time spent in a particular recursive call, labeled with the size of its subproblem.

Quick-Sort/Pivot-Sort

Divide and conquer.

1. Pick a pivot.
2. Divide into three parts: < pivot, = pivot, and > pivot (where hard work is done), can be done in-place to save space.
3. Recursively sort the < pivot part and the > pivot part.
4. Merge the sorted parts by concatenation (hard work is alrd done is step 2).

```
def inplace_quick_sort(S, a, b):
  """Sort the list from S[a] to S[b] inclusive using the quick-sort algorithm."""
  if a >= b: return                                  # range is trivially sorted
  pivot = S[b]                                       # last element of range is pivot
  left = a                                           # will scan rightward
  right = b−1                                        # will scan leftward
  while left <= right:
    # scan until reaching value equal or larger than pivot (or right marker)
    while left <= right and S[left] < pivot:
      left += 1
    # scan until reaching value equal or smaller than pivot (or left marker)
    while left <= right and pivot < S[right]:
      right −= 1
    if left <= right:                                # scans did not strictly cross
      S[left], S[right] = S[right], S[left]                          # swap values
      left, right = left + 1, right − 1                              # shrink range

  # put pivot into its final place (currently marked by left index)
  S[left], S[b] = S[b], S[left]
  # make recursive calls
  inplace_quick_sort(S, a, left − 1)
  inplace_quick_sort(S, left + 1, b)
```

Code Fragment 12.6: In-place quick-sort for a Python list S.

Figure 12.14: Divide step of in-place quick-sort, using index l as shorthand for identifier left, and index r as shorthand for identifier right. Index l scans the sequence from left to right, and index r scans the sequence from right to left. A swap is performed when l is at an element as large as the pivot and r is at an element as small as the pivot. A final swap with the pivot, in part (f), completes the divide step.

If pivot is randomized or divides the list into $1/2$, $1/2$ or $1/4$, $3/4$ at each step, the expected runtime is $O(n \log n)$. If pivot is badly chosen and the list is alrd sorted, runtime could be $O(n^2)$ (sub-list is $O(n)$ for each of the n divisions).

Figure 12.13: A visual time analysis of the quick-sort tree T. Each node is shown labeled with the size of its subproblem.

Quick-Sort is not ideal for short sequences. Hybrid approach: Keep dividing and once the sub-problem size < some threshold, say, 50, switch to using insertion-sort, which is good on short sequences.

Linear-Time Sorting

If the entries being sorted satisfy some additional constraints, we may avoid using comparisons and sort them in linear time.

Consider the problem of sorting a sequence of entries, each a key-value pair, where the keys have a restricted type.

Bucket-Sort

If we know that the range of the keys is, say, $[0, N-1]$ where N is a constant, then can sort in $O(n+N) = O(n)$ time.

Given a sequence S with n entries:

1. Create a bucket array B with N entries.
2. Remove each element k from S and insert it at the end of bucket $B[k]$ (stable sort) in case there are multiple entries with the same key ($O(n)$).
3. Iterate the bucket array B from front to back (so that the elements come out sorted), iterate each bucket from front to back (stable sort), remove each entry and insert it at the end of S ($O(N)$).

So if N is $O(n)$ (e.g., a constant), this is linear time.

Radix-Sort

Given a sequence S with n entries of key-value pairs, can sort in lexicographic order ($(k_1, v_1) < (k_2, v_2)$ if $k_1 < k_2$ or $k_1 = k_2$ and $v_1 < v_2$) in $O(n)$ time.

1. Use stable bucket-sort to sort on the second component v_i.
2. Use stable bucket-sort to sort on the first component k_i (stability here guarantees that the sorted second component remains sorted when the first component is the same).

Comparing Sorting Algorithms

Algorithm	Runtime	Stability	Use case
Selection-Sort	$O(n^2)$ best case	Can be stable.	
Insertion-Sort	$O(n^2)$ worst/average case, $O(n)$ best case	Stable.	Small sequences (<50)
Heap-Sort	$O(n \log n)$	Not stable.	Small sequences
Quick-Sort/Pivot-Sort	$O(n^2)$ worst case, $O(n \log n)$ average case	Not stable.	Large sequences
Merge-Sort	$O(n \log n)$ worst case	Stable.	Difficult to do in-place.
Bucket-Sort, Radix-Sort	$O(n)$	Stable.	Key range is known in advance.

Selection

The selection problem: Selecting the kth smallest element from an unsorted collection of n comparable elements.

Prune-and-Search/Decrease-and-Conquer

E.g., binary search.

Randomized Quick-Select

Similar to randomized quick-sort.

```
def quick_select(S, k):
  """Return the kth smallest element of list S, for k from 1 to len(S)."""
  if len(S) == 1:
    return S[0]
  pivot = random.choice(S)              # pick random pivot element from S
  L = [x for x in S if x < pivot]       # elements less than pivot
  E = [x for x in S if x == pivot]      # elements equal to pivot
  G = [x for x in S if pivot < x]       # elements greater than pivot
  if k <= len(L):
    return quick_select(L, k)           # kth smallest lies in L
  elif k <= len(L) + len(E):
    return pivot                        # kth smallest equal to pivot
  else:
    j = k − len(L) − len(E)             # new selection parameter
    return quick_select(G, j)           # kth smallest is jth in G
```

Code Fragment 12.9: Randomized quick-select algorithm.

$O(n)$ average case, $O(n^2)$ worst case.

Text-Processing

Pattern-Matching

Given a text string T of length n and a pattern string P of length m, find whether P is a substring of T.

Brute Force

For each of the $n - m + 1$ possible starting index of P, try matching each of the m characters in P.

$O(mn)$.

Boyer-Moore

Like brute force, but skip over indices that can't possibly match.

Worst case $O(mn)$.

Figure 13.2: A simple example demonstrating the intuition of the Boyer-Moore pattern-matching algorithm. The original comparison results in a mismatch with character e of the text. Because that character is nowhere in the pattern, the entire pattern is shifted beyond its location. The second comparison is also a mismatch, but the mismatched character s occurs elsewhere in the pattern. The pattern is next shifted so that its last occurrence of s is aligned with the corresponding s in the text. The remainder of the process is not illustrated in this figure.

Figure 13.3: Additional rules for the character-jump heuristic of the Boyer-Moore algorithm. We let i represent the index of the mismatched character in the text, k represent the corresponding index in the pattern, and j represent the index of the last occurrence of $T[i]$ within the pattern. We distinguish two cases: (a) $j < k$, in which case we shift the pattern by $k - j$ units, and thus, index i advances by $m - (j + 1)$ units; (b) $j > k$, in which case we shift the pattern by one unit, and index i advances by $m - k$ units.

```
def find_boyer_moore(T, P):
  """Return the lowest index of T at which substring P begins (or else -1)."""
  n, m = len(T), len(P)                    # introduce convenient notations
  if m == 0: return 0                      # trivial search for empty string
  last = { }                               # build 'last' dictionary
  for k in range(m):
    last[ P[k] ] = k                       # later occurrence overwrites
  # align end of pattern at index m-1 of text
  i = m−1                                  # an index into T
  k = m−1                                  # an index into P
  while i < n:
    if T[i] == P[k]:                       # a matching character
      if k == 0:
        return i                           # pattern begins at index i of text
      else:
        i −= 1                             # examine previous character
        k −= 1                             # of both T and P
    else:
      j = last.get(T[i], −1)               # last(T[i]) is -1 if not found
      i += m − min(k, j + 1)               # case analysis for jump step
      k = m − 1                            # restart at end of pattern
  return −1
```

Code Fragment 13.2: An implementation of the Boyer-Moore algorithm.

c	a	b	c	d
last(c)	4	5	3	-1

Figure 13.4: An illustration of the Boyer-Moore pattern-matching algorithm, including a summary of the last(c) function. The algorithm performs 13 character comparisons, which are indicated with numerical labels.

Knuth-Morris-Pratt (KMP)

Like Boyer-Moore, but use a failure table to make maximum skips upon mismatches.

$O(m+n)$.

Figure 13.5: A motivating example for the Knuth-Morris-Pratt algorithm. If a mismatch occurs at the indicated location, the pattern could be shifted to the second alignment, without explicit need to recheck the partial match with the prefix `ama`. If the mismatched character is not an `l`, then the next potential alignment of the pattern can take advantage of the common `a`.

Example 13.2: *Consider the pattern $P =$ `"amalgamation"` from Figure 13.5. The Knuth-Morris-Pratt (KMP) failure function, $f(k)$, for the string P is as shown in the following table:*

k	0	1	2	3	4	5	6	7	8	9	10	11
$P[k]$	a	m	a	l	g	a	m	a	t	i	o	n
$f(k)$	0	0	1	0	0	1	2	3	0	0	0	0

```
def find_kmp(T, P):
  """Return the lowest index of T at which substring P begins (or else -1)."""
  n, m = len(T), len(P)                    # introduce convenient notations
  if m == 0: return 0                      # trivial search for empty string
  fail = compute_kmp_fail(P)               # rely on utility to precompute
  j = 0                                    # index into text
  k = 0                                    # index into pattern
  while j < n:
    if T[j] == P[k]:                       # P[0:1+k] matched thus far
      if k == m − 1:                       # match is complete
        return j − m + 1
      j += 1                               # try to extend match
      k += 1
    elif k > 0:
      k = fail[k−1]                        # reuse suffix of P[0:k]
    else:
      j += 1
  return −1                                # reached end without match
```

Code Fragment 13.3: An implementation of the KMP pattern-matching algorithm. The compute_kmp_fail utility function is given in Code Fragment 13.4.

```
def compute_kmp_fail(P):
  """Utility that computes and returns KMP 'fail' list."""
  m = len(P)
  fail = [0] * m                 # by default, presume overlap of 0 everywhere
  j = 1
  k = 0
  while j < m:                   # compute f(j) during this pass, if nonzero
    if P[j] == P[k]:             # k + 1 characters match thus far
      fail[j] = k + 1
      j += 1
      k += 1
    elif k > 0:                  # k follows a matching prefix
      k = fail[k−1]
    else:                        # no match found starting at j
      j += 1
  return fail
```

Code Fragment 13.4: An implementation of the compute_kmp_fail utility in support of the KMP pattern-matching algorithm. Note how the algorithm uses the previous values of the failure function to efficiently compute new values.

Dynamic Programming

For finding optimal solutions.

Knapsack Problem

Given n items, their values `values` and weights `weights`, and a knapsack with weight capacity `W`, how to pick items to put in the knapsack such that their total value is highest?

Brute Force

For each item, it is either included in the knapsack or excluded. $O(2^n)$.

```
def knapSack(W, weights, values, n):

    if n == 0 or W == 0:
        return 0
    if (weights[n-1] > W): # item n is heavier than knapsack capacity, so can't
include it
        return knapSack(W, weights, values, n-1) # exclude item n
    else:
        return max(
            values[n-1] + knapSack(
                W-weights[n-1], weights, values, n-1), # include item n
            knapSack(W, weights, values, n-1) # exclude item n
            )
```

Memoisation

We implement memoization by computing, bottom-up, a table `M` as a two-dimensional array, where `M[i][w]` stores the maximal price achievable by taking only the first `i` items while not exceeding the weight `W`. This improves time complexity to $O(n \cdot W)$ by replacing the two recursive calls with two lookups in the memoization table.

```
def knapSack(W, weights, values, n):
    M = [[0 for x in range(W + 1)] for x in range(n + 1)] # initialize
memoization table

    # build memoization table from bottom-up
    for i in range(n + 1): # row is the range of indices of items allowed to
take (from 0 to n)
        for w in range(W + 1): # column is weight limit (from 0 to W)
            if i == 0 or w == 0: # if only the first 0 item is allowed to take
or if weight limit is 0, maximal price is 0
                M[i][w] = 0
            elif weights[i-1] <= w: # if item's weight <= current weight limit
                M[i][w] = max( # recursive calls replaced by lookups in the
memoization table M
                            values[i-1] + M[i-1][w - weights[i-1]], # take this
item
                            M[i-1][w] # not taking this item
                            )
```

```
            else: # if item's weight > current weight limit, update the
memoization table M[i][w] to be the same as M[i-1][w] because there's no
improvement in price
                M[i][w] = M[i-1][w]

    return M[n][w] # return the maximal price achievable by taking all n items
while not exceeding the weight limit W
```

Reconstructing the list of items from the memoization table:

```
i  item    w  p |  0  1  2  3  4
-------------------------------
0  apple   1  1 |  0  1  1  1  1
1  melon   2  2 |  0  1  2  3  3
2  kiwi    1  2 |  0  2  3  4  5
3  durian  2  3 |  0  2  3  5  6
```

Let `sack = []` . We start from the rightmost, bottom-most cell: `weight=4` and `i=3`, with price `6`.

- The previous row has price `5`, less than `6` in current row, so durian was taken, and `sack = [3]`.
- We subtract Durian's weight `2` and go to column `4 - 2 = 2` in the same row, repeating the process. The previous row has price `2`, less than `3` in current row, so kiwi was taken, and `sack = [3, 2]`.
- We subtract Kiwi's weight `1` and go to column `2 - 1 = 1` in the same row. The previous row has price `1`, same as current row, so melon was not taken.
- Finally, apple was taken, so `sack = [3, 2, 0]`.

Graph

Graph. A graph G is a set V of vertices and a collection E of pairs of vertices from V, called edges.

- A way of representing relations among objects in V.

Path. A path is a sequence of alternating vertices and edges that starts at a vertex and ends at a vertex such that each edge is incident to its predecessor and successor vertex.

Cycle. A cycle is a path that starts and ends at the same vertex, and that includes at least one edge.

Simple path. A path is simple if each vertex in the path is distinct.

Simple cycle. A cycle is simple if each vertex in the cycle is distinct, except for the first and last one.

Spanning subgraph. A spanning subgraph of G is a subgraph of G that contains all the vertices of the graph G.

Graph Traversal

Depth-First Search

Algorithm DFS(G,u): {We assume u has already been marked as visited}

Input: A graph G and a vertex u of G

Output: A collection of vertices reachable from u, with their discovery edges

for each outgoing edge e = (u,v) of u **do**
  if vertex v has not been visited **then**
    Mark vertex v as visited (via edge e).
    Recursively call DFS(G,v).

Code Fragment 14.4: The DFS algorithm.

```
def DFS(g, u, discovered):
  """Perform DFS of the undiscovered portion of Graph g starting at Vertex u.

  discovered is a dictionary mapping each vertex to the edge that was used to
  discover it during the DFS. (u should be "discovered" prior to the call.)
  Newly discovered vertices will be added to the dictionary as a result.
  """
  for e in g.incident_edges(u):          # for every outgoing edge from u
    v = e.opposite(u)
    if v not in discovered:              # v is an unvisited vertex
      discovered[v] = e                  # e is the tree edge that discovered v
      DFS(g, v, discovered)              # recursively explore from v
```

Code Fragment 14.5: Recursive implementation of depth-first search on a graph, starting at a designated vertex *u*.