Algorithms and Datastructures

December 24, 2023

Contents

List of Algorithms

1 Search Algorithms

Input

- Array of elements
- Value to find

Output

- Index of element
- Or; No such Element

1.1 Linear Search

Pro

- Elements don't have to be sorted

Con

- Slow runtime

Runtime $O(n)$

Algorithm 1 LinearSearch(A, b)

```
for i = 1, ..., n do
    if A[i] = b then
        return i
    end if
end for
return Not Found
```

1.2 Binary Search

Pro

- Fast runtime

Con

- Elements have to be sorted

Runtime $O(\log n)$

Algorithm 2 BinarySearch(A, b)

```
l ← 0, r ← n
while l ≤ r do
    m ← l + ⌊(r−l)/2⌋
    if A[m] = b then
        return m
    end if
    if b < A[m] then
        l ← m − 1
    else
        r ← m + 1
    end if
end while
```

2 Sorting Algorithms

Input
- Array of elements

Output
- Sorted Array ($A[1] \leq A[2] \leq \cdots \leq A[n]$)

2.1 Bubble Sort

Pro
- Easy implemented

Con
- Horrible runtime, never actually use this.

Runtime $O(n^2)$

Algorithm 3 BubbleSort(A)

```
1: for i = 1, ..., n do
2:     for j = 1, ..., n − 1 do
3:         if A[j] > A[j + 1] then
4:             switch A[j] and A[j + 1]
5:         end if
6:     end for
7: end for
```

2.2 Selection Sort

Pro
- Easy implemented
- Less switching of elements then Bubble Sort (only $O(n)$)

Con
- Still horrible runtime

Runtime $O(n^2)$

Algorithm 4 SelectionSort(A)

```
1: for j = 1, ..., n do
2:     k ← 0
3:     for i = 1, ..., j do                ▷ assigns the maximum value in A[1, ..., j] to k
4:         if A[i] > A[k] then
5:             k ← i
6:         end if
7:     end forswitch A[k] and A[n − j + 1]
8: end for
```

2.3 InsertionSort

Pro
- Easy implemented
- Less comparissons then Bubble Sort (only $O(n \log n)$)

Con

- Still horrible runtime

Runtime $O(n^2)$

Algorithm 5 InsertionSort(A)

for $j = 2, \ldots, n$ **do**
    find position k of $A[j]$ in $A[1, \ldots, j-1]$
    insert $A[j]$ at $A[k]$ and move $A[k, \ldots, j-1]$ to $A[k+1, \ldots, j]$
end for

2.4 Merge Sort

Pro

- Good Runtime

Con

- Complicated to implement (especially ecause you need to implement the merge)

Runtime $O(n \log n)$

Algorithm 6 Merge(A, l, m, r)

$B \leftarrow [1, \ldots, r-l+1]$
$i \leftarrow l$
$j \leftarrow m+1$
$k \leftarrow 1$
while $i \leq m \wedge j \leq r$ **do**
    if $A[i] < A[j]$ **then**
        $B[k] \leftarrow A[i]$
        $i \leftarrow i+1$, $k \leftarrow k+1$
    else
        $B[k] \leftarrow A[j]$
        $j \leftarrow j+1$, $k \leftarrow k+1$
    end if
end while
while $i \leq m$ **do**
    $B[k] \leftarrow A[i]$
    $i \leftarrow i+1$, $k \leftarrow k+1$
end while
while $j \leq r$ **do**
    $B[k] \leftarrow A[j]$
    $j \leftarrow j+1$, $k \leftarrow k+1$
end while
copy B to $A[l, \ldots, r]$

Algorithm 7 MergeSort($A, l = 0, r = n$)

if $f < r$ **then**
    $m \leftarrow l + \lfloor \frac{r-l}{2} \rfloor$
    MergeSort(A,l,m)
    MergeSort(A,m+1,r)
    Merge(A,l,m,r)
end if

2.5 Quick Sort

Pro

- Runtime almost as good,of a runtime as merge sort, but less space complexity.

Con

- Not easy to implement
- Can take $O(n^2)$

Runtime $O(n^2)$ (worst-case) $O(n \log n)$ (most cases)

Algorithm 8 QuickSort(A, l, r)

```
1: if l < r then
2:     k ← Split(A, l, r)
3:     QuickSort(A,l,k-1)
4:     QuickSort(A,k+1,r)
5: end if
```

Algorithm 9 Split(A, l, r)

```
1: i ← l, j ← r − 1, p ← A[r]
2: repeat
3:     while i < r ∧ A[i] ≤ p do
4:         i ← i + 1
5:     end while
6:     while j ≥ l ∧ A[i] > p do
7:         j ← j + 1
8:     end while
9:     if i < j then
10:        switch A[i] and A[j]
11:    end if
12: until i > j
13: switch A[i] and A[j]
```

2.6 Heap Sort

Pro

- Good Runtime

Con

- Requires extra datastructure and space, refere to section 3.5.1
- Bad Locality

Runtime $O(n \log n)$

Algorithm 10 HeapSort(A)

```
1: H ← make-heap(A)
2: for i = 1, ..., n do
3:     A[i] ← extract-min(H)
4: end for
```

3 Datastructures and Abstract Data Types (ADTs)

3.1 Dictionary/Map

maps a key to a value

3.1.1 Functions

Function	Description
insert(k, v)	maps the key k to the value v.
remove(k)	unmaps the key k.
get(k)	gets the associated value of the key k.

3.1.2 Implementation comparisson

	AVL Tree
insert(k, v)	$O(\log n)$
remove(k)	$O(\log n)$
get(k)	$O(\log n)$

3.2 List

A ordered list of elements, can be indexed.

3.2.1 Functions

Function	Description
addFirst(v)	adds the element v to the beginning of the list.
addLast(v)	adds the element v to the end of the list.
insert(i, v)	sets the value at index i to v.
remove(v)	removes the element v from the list.
removeFirst()	removes the first element from the list.
removeLast()	removes the last element from the list.
insertAfter(v, v')	insertst the element v' after the element v.
get(i)	gets the element at index i.

3.2.2 Implementation comparisson

	ArrayList	LinkedList	DoublyLinkedList
Size in memory	n·tSize	n(tSize+addrSize)	n(tSize+2·addrSize)
Locality	great	fragmented	fragmented
addFirst(v)	$O(\alpha(n))$	$O(1)$	$O(1)$
addLast(v)	$O(\alpha(1))$	$O(1)$[1]	$O(1)$[1]
insert(i, v)	$O(\alpha(n))$	$O(n)$	$O(n)$
insertAfter(v, v')	$O(n)$	$O(1)$	$O(1)$
remove(v)	$O(n)$	$O(n)$	$O(1)$
removeFirst()	$O(n)$	$O(1)$	$O(1)$
removeLast()	$O(1)$	$O(n)$	$O(1)$
get(i)	$O(1)$	$O(n)$	$O(n)$

[1] If last element is known, otherwhise $O(n)$

3.3 Queue

First in first out.

3.3.1 Function

Function	Description
enqueue(v)	adds the element v to the queue.
dequeue()	removes the element that has been in the queue for the longest.

3.3.2 Implementation comparisson

	(Doubly)LinkedList
enqueue(v)	$O(1)$
dequeue()	$O(1)$

3.4 Stack

Last in first out.

3.4.1 Function

Function	Description
push(v)	adds the element v to the stack.
pop()	removes the element on top of the stack (the one added the most recent).

3.4.2 Implementation comparisson

	(Doubly)LinkedList
push(v)	$O(1)$
pop()	$O(1)$

3.5 Sorted Datastructures

If we regularly want to search in a mutable set of data, we may want to store it in a way that we can easily search through it. In this section we disscuss such datastructures.

3.5.1 Binary Tree/Heap

A binary tree is a tree where every node has two children, except the ones in the last two rows. The last row fills from left to right. Always has the element that satisfies the heap condition the best on the top. After every change the heap condition has to be restored with the following algorithm.

Algorithm 11 make-heap(A)

1: **for** $k = \lfloor \frac{A.\text{length}}{2} \rfloor, \ldots, 1$ **do**
2: heapify(A, k)
3: **end for**

Algorithm 12 extract(-min)(H)

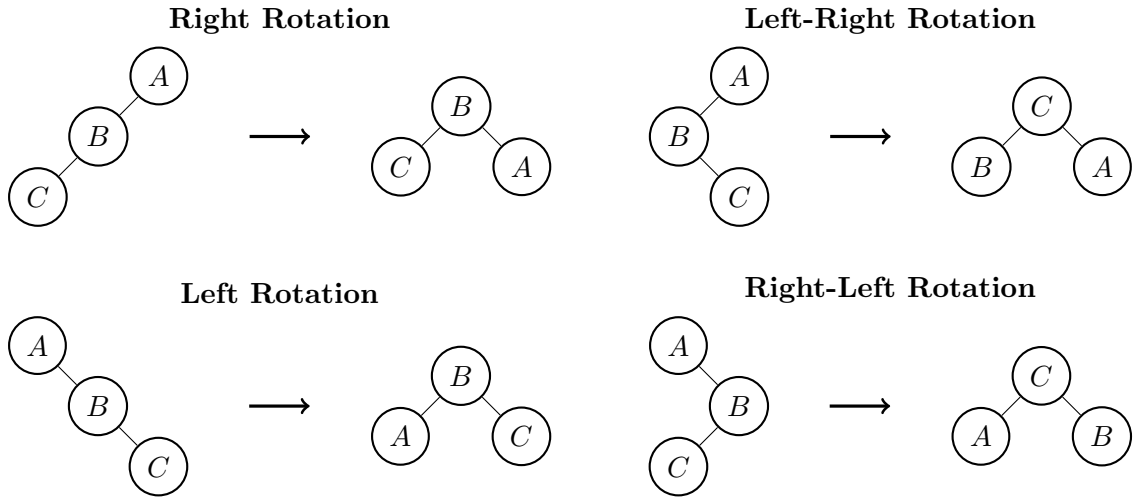
1: res$\leftarrow H[0]$
2: $H[0] \leftarrow H[H.\text{length}]$
3: $H.\text{length} \leftarrow H.\text{length} - 1$
4: **return** res

Algorithm 13 (min-)heapify(A, s)

```
1: left ← 2s
2: right ← 2s + 1
3: if left ≤ A.length ∧ A[left] < A[s] then
4:     smallest← left
5: else
6:     smallest← s
7: end if
8: if right ≤ A.length ∧ A[right] < A[smallest] then
9:     smallest ← right
10: end if
11: if smallest≠s then
12:     swap(A[s], A[smallest])
13:     heapify(A, smallest)
14: end if
```

3.5.2 AVL Tree

A AVL tree is a datastructure where searching for a element is possible in $O(\log n)$. The tree has a balance factor $|h_l - h_r|$ and if it's absolute value gets bigger then 1 at A, depending on where the imbalance at came from we do one of the following rotations (C is the vertex from where the imbalance came from). You have to apply this process from the bottom up for it to work.



3.6 Union Find

Algorithm 14 make(V)

```
1: rep[v] ← v ∀ ∈ V
```

Algorithm 15 same(u, v)

```
1: return rep[u] = rep[v]
```

Algorithm 16 union(u, v)

```
1: for x ∈ members[rep[u]] do
2:     rep[x] ← rep[v]
3:     members[rep[v]] ← members[rep[v]] ∪ {x}
4: end for
```

4 Dynamic Programming

An approach to problem solving by spliting the problem into smaller subproblems that can be solved recursivly. Often this recursion gets represented as a table of dimension of the recursion inputs. The dimensions often can be read of the runtime unless one iteration has a non constant runtime.

5 Graphs

Graph Theory is an abstraction layer to a lot of very common problems in computer science. By abstracting these problems to graphs we can apply lots of graph algorithms to easily solve these problems. A graph can be used to represent networks, relations, etc.

5.1 Terms and Definitions

Term	Description	Mathematical Definition
Graph G	a Graph is a set of vertices and a set of edges.	$G = (V, E)$
Edge e	a edge e connects two vertices u and v.	$e = \{u, v\}$
u adjacent[2] to v	there exists an edge between u and v.	$\{u, v\} \in E$
e incident to v	e is a edge between v and any other vertex.	$\exists u$ where $e = \{u, v\} \in E$
Degree of v	the amount of incident edges to v.	$\deg(v) = \lvert\{e \in E \lvert \exists u, e = u, v\}\rvert$
Walk W	a sequence W of directly adjacent vertices of length l.	$W = (v_0, \ldots, v_l)$
Path P	walk P with length l and no repeating vertices.	$P = (v_0, \ldots, v_l)$ where $\forall i, j\, 0 \leq i, j \leq l, v_i \neq v_i$
Closed Walk Z	a walk Z with length l and the first and the last vertices being the same.	$Z = (v_0, \ldots, v_l)$ where $v_0 = v_l$
Cycle C	a walk C with length l and none except for the first and the last vertices not being the same.	$C = (v_0, \ldots, v_l)$ where $\forall i, j\, 0 < i, j < l, v_i \neq v_j$ and $v_0 = v_l$
v reachable[3] from u	there exists a walk from u to v (equivalence relation)	$\exists C$ where $C = (u, \ldots, v)$
connected component	equivalence class of reachable.	
connected graph	if there exists only one connected component in a graph.	$\forall u, v \in V, v$ reachable form u
Euler walk	a walk which contains every edge exactly once.	
Hamilton path	a path which contains every vertex exactly once.	
Euler circuit[4]	a closed walk which contains every edge exactly once.	

[2] sometimes directly adjacent, since there also exists indirectly adjacent
[3] sometimes indirectly adjacent to instead of reachable from
[4] sometimes also called euler cycle or euler tour

5.2 EulerTour Algorithm

Finds a Euler tour

Runtime $O(m)$

Algorithm 17 EulerTour(G)

```
1: Z ← empty list
2: for each v ∈ V, unmarked do
3:     EulerWalk(G, Z, v)
4: end for
5: return Z
```

Algorithm 18 EulerWalk(G, Z, u)

```
1: for {u, v} ∈ E unmarked do
2:     mark {u, v}
3:     EulerWalk(G, Z, v)
4: end for
```

5.3 Directed Graph; Terms and Definitions

Term	Description	Mathematical Definition
Directed Graph	Same as a undirected graph, except for edges, now they have a direction.	$G = (V, E)$
(directed) Edge e	a edge from vertex u to vertex v.	$e = (u, v)$
v successor to u	there is an edge from u to v.	$(u, v) \in E$
u predeccessor to v	there is an edge from u to v.	$(u, v) \in E$
in-degree of v	the number of edges to v.	$\deg_{\text{in}}(v) = \lvert\{u \in V \lvert (u, v) \in E\}\rvert$
out-degree of u	the number of edges from u.	$\deg_{\text{out}}(u) = \lvert\{v \in V \lvert (u, v) \in E\}\rvert$
source u	a vertex with no edges to itself.	$\deg_{\text{in}} = 0$
sink v	a vertex with no edges from itself.	$\deg_{\text{out}} = 0$

5.4 Depth-First-Search (DFS)

Goes through all the vertices, while first going as deep as possible and then exploring other options.

Runtime $O(n + m)$ with adjacency-lists, $O(n^2)$ with adjacency-matrices

Algorithm 19 DFS(G)

```
1: for u ∈ V, unmarked do
2:     visit(u)
3: end for
```

Algorithm 20 visit(G, u)

```
1: mark u
2: for {u, v} ∈ E, where v unmarked do
3:     visit(G, v)
4: end for
```

5.5 Breadth-First-Search

Goes through all the vertices, while first exploring all options one level deep, then doing the same on the next level of deepness.

Runtime $O(n+m)$ with adjacency-lists, $O(n^2)$ with adjacency-matrices

Algorithm 21 BFS(G, s)

```
1: Q ← {s}
2: while Q ≠ ∅ do
3:     u ← dequeue(Q)
4:     for (u, v) ∈ E, where v unmarked do
5:         mark v
6:         enqueue(Q, v)
7:     end for
8: end while
```

5.6 Shortest Path Algorithms

5.6.1 Dijkstra

Finds the shortest path from vertex s to all other vertices on a graph without negative weighted edges.

Runtime $O((m+n) \cdot \log n)$

Algorithm 22 Dijkstra(G, s)

```
1: d[s] ← 0;   d[v] ← ∞ ∀v ∈ V\{s}
2: H ← make-heap(V); decrease-key(H, s, 0)
3: while V not empty do
4:     u ← extract-min(H)
5:     mark u
6:     for (u, v) ∈ E, v unmarked do
7:         d[v] ← min{d[v], d[u] + c(u, v)}
8:         decrease-key(H, v, d[v])
9:     end for
10: end while
```

5.6.2 Bellman-Ford

Finds the shortest path from vertex s to all other vertices on a graph with non-negative and negative weighted edges.

Algorithm 23 Bellman-Ford(G, s)

```
1: d[s] ← 0;   d[v] ← ∞ ∀v ∈ V\{s}
2: for i ∈ {1, ..., n − 1} do
3:     for (u, v) ∈ E do
4:         d[v] ← min{d[v], d[u] + c(u, v)}
5:     end for
6: end for
```

5.7 MST Algorithms

5.7.1 Boruvka

Runtime $O((m+m)\cdot \log n)$

Algorithm 24 Boruvka(G)

```
F ← ∅
while F not a spanning tree do
    (S_1, ..., S_k) ← connected components in the graph G' = (V, F)
    (e_1, ..., e_k) ← minimal edges on S_1, ..., S_k
    F ← F ∪ {e_1, ..., e_k}
end while
```

5.7.2 Prim

Runtime $O((m+n)\cdot \log n)$

Algorithm 25 Prim(G, s)

```
H ← make-heap(V, ∞)
d[s] ← 0; d[v] ← ∞ ∀v ∈ V\{s}
decrease-key(H, s, 0)
while H ≠ ∅ do
    u ← extract-min(H)
    mark u
    for {u, v} ∈ E, v unmarked do
        d[v] ← min{d[v], c(u, v)}
        decrease-key(H, v, d[v])
    end for
end while
```

5.7.3 Kruskal

Runtime $O(m\cdot \log m + n\cdot \log n)$

Algorithm 26 Kruskal(G)

```
F ← ∅
UF ← make-union-find(V)
sort(E)
for {u, v} ∈ E do
    if ¬same(u, v) then
        F ← F ∪ {{u, v}}
        union(u, v)
    end if
end for
```

5.8 All to all shortest path Algorithms

5.8.1 Floyd-Warshall Algorithm

Runtime $O(n^3)$

Algorithm 27 Floyd-Warshall(G)

for $u \in V$ **do**
  $d^0_{uu} \leftarrow 0$
  for $v \in V \backslash \{u\}$ **do**
    if $(u, v) \in E$ **then**
      $d^0_{uv} \leftarrow c(u, v)$
    else
      $d^0_{uv} \leftarrow \infty$
    end if
  end for
end for
for $i = 1, \ldots, n$ **do**
  for $u \in V$ **do**
    for $v \in V$ **do**
      $d^i_{uv} \leftarrow \min\{d^{i-1}_{uv}, d^{i-1}_{ui} + d^{i-1}_{iv}\}$
    end for
  end for
end for
return d^n

5.8.2 Johnson's Algorithm

Runtime $O(n \cdot (m + n) \cdot \log n)$

Algorithm 28 Johnson(G)

$G' = (V', E')$ where $V' \leftarrow V \cup \{z\}, E' \leftarrow E \cup \{\{z, v_0\}, \ldots, \{z, v_n\}\}$
$h \leftarrow$ Bellman-Ford(G', z)
$\hat{c}(u, v) \leftarrow c(u, v) + h(u) - h(v)$
$d[v] \leftarrow$ Dijkstra(G with the cost function $\hat{c}, s$) $\forall v \in V$
return d