Sim's Computing Revision Notes

http://simshadows.com/
contact@simshadows.com

Last Updated: April 29, 2023

Contents

Chapter 1

Fundamentals

1.1 Complexity, Big O, And Other Performance Topics

TODO: Write this section!

TODO: Also talk about complexity classes, NP completeness, and the P=NP problem.

1.2 Binary Representation and Arithmetic

1.2.1 Introduction

TODO: Write this?

TODO: Make sure you cover:

- bit shifting
- IEEE 754 floating point, including special values, and cover quantization error and other important implications
- overcoming overflow/underflow
- ASCII, Unicode, and possibly other character encodings
- Modular arithmetic (especially useful for competitive programming)

1.3 Dynamic Arrays

TODO: Write this section!

1.4 Array Search, Selection, and Partitioning Algorithms

TODO: Write this section!

TODO: Include Quickselect.

1.5 Sorting Algorithms

TODO: Write this section!

1.6 Linked Lists

TODO: Write this section!

1.7 Binary Search Trees (BSTs)

TODO: Write this section!

1.8 Binary Heaps

1.8.1 Introduction

A *heap* is a tree data structure where a *heap property* is fulfilled:

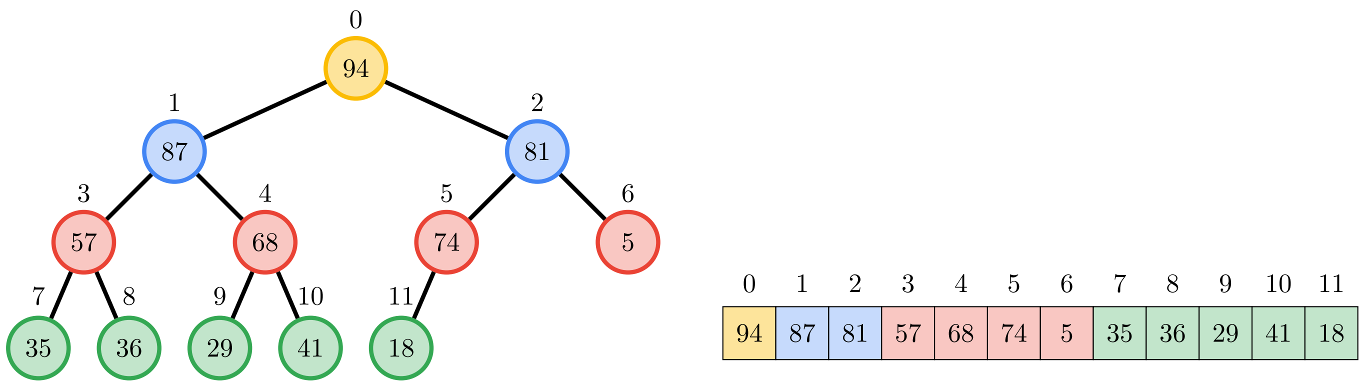
- For a *max-heap*, every node's value is greater than or equal to each of their immediate children.
- For a *min-heap*, every node's value is lesser than or equal to their immediate children.

A *binary heap* is a heap with the form of a binary tree that also fulfills the *shape property*, requiring the tree to also form a *complete binary tree*, meaning:

1. all levels except the last are completely filled, and
2. the last level is filled from left to right.

Heaps are commonly used to implement the *priorityqueue* ADT.

Binary heaps are commonly efficiently implemented using an array (see Fig. 1.1(b)). Notably, each row appears in the array in order from left to right. Since the binary heap is a complete binary tree, the array is a compact representation with no missing nodes before the last node.



(a) The conceptual tree structure of the heap.

(b) Implementation using the array representation.

Figure 1.1: Example of a *binary max-heap* data structure. (Colours are used only to help show structure.)

Table 1.1: Some of the most common operations of a binary heap.

Operation	**Runtime**	**Basic Summary**
Push(*item*)	$O(\log n)$	Add a single item.
Pop() $\rightarrow$ *item*	$O(\log n)$	Extract and remove the root of the heap (i.e. the biggest item of a max-heap, or the smallest item of a min-heap).
Push-Pop(*item*) $\rightarrow$ *item*	$O(\log n)$	Equivalent to a push and then a pop, in that order. *Typically more efficient than using the individual operations.*
Pop-Push(*item*) $\rightarrow$ *item*	$O(\log n)$	Equivalent to a pop and then a push, in that order. *Typically more efficient than using the individual operations.*
Heapify()	$O(n)$	Convert an arbitrary *complete binary tree* into a valid heap.
Get Size() $\rightarrow$ *size*	$O(1)$	Get the number of items in the heap.
(Internal) Sift-Up(*node*)	$O(\log n)$	Moves a node up the tree until it is in a valid position.
(Internal) Sift-Down(*node*)	$O(\log n)$	Moves a node down the tree until it is in a valid position.

TODO: List out some more operations?

1.8.2 Array Indexing

For 0-indexed arrays, given an index i, we can find the indices of its left child c_1, right child c_2, and parent p with:

$$\boxed{c_1 = 2i + 1 \qquad c_2 = c_1 + 1 \qquad p = \left\lfloor \frac{i-1}{2} \right\rfloor}$$

To quickly derive these equations, let's try using a 1-indexed array:

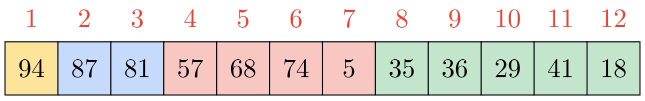


Figure 1.2: Array implementation of Fig. 1.1 with a 1-indexed array.

With 1-indexing, the pattern is much more obvious. Using i', c'_1, c'_2, and p' as our 1-indexed array indices, we observe that the following are true:

$$c'_1 = 2i' \qquad c'_2 = c'_1 + 1 \qquad p' = \left\lfloor \frac{i'}{2} \right\rfloor$$

To convert back to 0-indexing, we apply $i' = i + 1$ and such:

$$\begin{aligned} (c_1 + 1) &= 2(i+1) \\ c_1 + 1 &= 2i + 2 \\ c_1 &= 2i + 1 \end{aligned} \qquad \begin{aligned} (c_2 + \cancel{1}) &= (c_1 + \cancel{1}) + 1 \\ c_2 &= c_1 + 1 \end{aligned} \qquad \begin{aligned} p + 1 &= \left\lfloor \frac{i+1}{2} \right\rfloor \\ p &= \left\lfloor \frac{i+1-2}{2} \right\rfloor = \left\lfloor \frac{i-1}{2} \right\rfloor \end{aligned}$$

TODO: Is there a good intuition behind these equations?

TODO: Discuss left/right-shift's role in implementing array indexing.

1.8.3 Push and Sift-Up

Push(*item*) Sift-Up(*node*)

The *push* operation inserts a new item into the heap. The algorithm:

1. Add the new item to the next free space while still forming a complete binary tree.
2. Run *sift-up* starting from the new node to restore the heap property.

The *sift-up* operation moves a node up the tree until it is in the correct position. The algorithm:

1. Compare the current node with its parent. If they are in the correct order, then we are done.
2. Otherwise, swap them, move to the new position of our node, and repeat *sift-up*.

Both algorithms have $O(\log n)$ runtime.

Sample implementation:

Listing 1.1: Sample Python implementation for *push* and *sift-up* operations on a max-heap, implemented using a list.

```python
def hpush(heap, item):
    heap.append(item)
    sift_up(heap, len(heap) - 1)

def sift_up(heap, i):
    parent = (i - 1) >> 1
    if (i == 0) or (heap[parent] > heap[i]):
        return
    (heap[parent], heap[i]) = (heap[i], heap[parent])
    sift_up(heap, parent)
```

Listing 1.2: Sample driver code for Listing 1.1.

```python
# Start with a valid heap
heap = [94, 87, 81, 57, 68, 74, 5, 35, 36, 29, 41, 18]

# We push 92 to the heap
hpush(heap, 92)
print(heap) # [94, 87, 92, 57, 68, 81, 5,
            #  35, 36, 29, 41, 18, 74]
```

TODO: Provide a visual explainer of what's happening?

1.8.4 Pop and Sift-Down

$$\text{Pop}() \to \textit{item} \qquad\qquad \text{Sift-Down}(\textit{node})$$

The *pop* operation extracts and removes the root of the heap. For a max-heap, the extracted value is the largest item, or for a min-heap, the extracted value is the smallest item. The algorithm:

1. Store the current root item. This will be returned later.
2. Move the bottom-leftmost item into the original place of root.
3. Run *sift-down* starting from the new root to restore the heap property.

The *sift-down* operation moves a node down the tree until it is in the correct position. The algorithm:

1. If our current node is in the correct order with its children, then we are done.
2. Otherwise, swap the current node and one of the children it's out-of-order with. For a max-heap, we must swap with the larger child, and for a min-heap, we must swap with the smaller child, otherwise these nodes will still be out of order.
3. Move to our node's new location, and repeat *sift-down*.

Both algorithms have $O(\log n)$ runtime.

Sample implementation:

Listing 1.3: Sample Python implementation for *pop* and *sift-down* operations on a max-heap, implemented using a list.

```
# PRECONDITION: len(heap) > 0
def hpop(heap):
    to_return = heap[0]
    heap[0] = heap.pop()
    sift_down(heap, 0)
    return to_return

def sift_down(heap, i):
    left = (i << 1) + 1
    if left >= len(heap):
        return
    right = left + 1

    if (right >= len(heap)) \
            or (heap[left] > heap[right]):
        next_child = left
    else:
        next_child = right

    if heap[next_child] <= heap[i]:
        return
    (heap[next_child], heap[i]) = \
            (heap[i], heap[next_child])
    sift_down(heap, next_child)
```

Listing 1.4: Sample driver code for Listing 1.3.

```
# Start with a valid heap
heap = [94, 87, 81, 57, 68, 74, 5, 35, 36, 29, 41, 18]

result = hpop(heap)
print(result) # 94
print(heap) # [87, 68, 81, 57, 41, 74,
            #  5, 35, 36, 29, 18]
```

1.8.5 Push-Pop and Pop-Push

$$\text{Push-Pop}(\textit{item}) \to \textit{item} \qquad\qquad \text{Pop-Push}(\textit{item}) \to \textit{item}$$

The *push-pop* operation is equivalent to a *push* and then a *pop*, in that order. The algorithm:

1. If our new item is greater than or equal to the current root (for a max-heap) or less than or equal to the current root (for a min-heap), then we are done. No need to do any further operations on the heap. The new item is also the popped item.
2. Store the current root. This will be returned later.
3. Replace the root node with the new item.
4. Perform *sift-down* starting from the root node.

The *pop-push* operation is the other way around (a *pop* followed by a *push*). The algorithm:

1. Store the current root. This will be returned later.
2. Replace the root node with the new item.
3. Perform *sift-down* starting from the root node.

Push-pop and *pop-push* are typically more efficient than using the individual *push* and *pop* operations together. Both operations have $O(\log n)$ runtime.

Sample implementation:

Listing 1.5: Sample Python implementation and driver code for *pop-push* on a max-heap, implemented using a list. For `sift_down`, refer to Listing 1.3.

```python
# PRECONDITION: len(heap) > 0
def hpushpop(heap, item):
    if item < heap[0]:
        (item, heap[0]) = (heap[0], item)
        sift_down(heap, 0)
    return item

# sift_down is defined elsewhere.

# A valid heap
template = [94, 87, 81, 57, 68, 74, \
            5, 35, 36, 29, 41, 18]

heap = template.copy()
result = hpushpop(heap, 37)
print(result) # 94
print(heap) # [87, 68, 81, 57, 41, 74,
            #  5, 35, 36, 29, 37, 18]

heap = template.copy()
result = hpushpop(heap, 95)
print(result) # 95
print(heap) # [94, 87, 81, 57, 68, 74, \
            #  5, 35, 36, 29, 41, 18]
```

Listing 1.6: Similar to Listing 1.5, but for *pop-push*.

```python
# PRECONDITION: len(heap) > 0
def hpoppush(heap, item):
    (item, heap[0]) = (heap[0], item)
    sift_down(heap, 0)
    return item


# sift_down is defined elsewhere.

# The same valid heap
template = [94, 87, 81, 57, 68, 74, \
            5, 35, 36, 29, 41, 18]

heap = template.copy()
result = hpoppush(heap, 37)
print(result) # 94
print(heap) # Same as for hpushpop(heap, 37)


heap = template.copy()
result = hpoppush(heap, 95)
print(result) # 94
print(heap) # [95, 87, 81, 57, 68, 74,
            #  5, 35, 36, 29, 41, 18]
```

TODO: Provide a visual explainer of what's happening?

1.8.6 Heapify

Heapify()

The *heapify* operation converts an arbitrary *complete binary tree* into a valid heap. The algorithm simply runs *sift-down* on array nodes $[\lfloor n/2 \rfloor, \ldots, 1, 0]$, in that order.

This algorithm runs in $O(n)$.

Sample implementation:

Listing 1.7: Sample Python implementation for *heapify* for max-heaps, using list-based complete binary trees.

```python
def heapify(heap):
    for i in range(len(heap) >> 1, 0, -1):
        sift_down(heap, i - 1)
```

Listing 1.8: Sample driver code for Listing 1.7.

```python
# Start with an arbitrary complete binary tree
heap = [18, 35, 74, 36, 29, 81, 5, 57, 87, 68, 41, 94]

heapify(heap)
print(heap) # [94, 87, 81, 57, 68, 74,
            #  5, 35, 36, 29, 41, 18]
```

TODO: Provide a visual explainer of what's happening?

TODO: Explain the naive algorithm of repeated push operations, and another alternative algorithm that uses sift-up. Explain why sift-down is the best algorithm.

1.8.7 Further Reading

TODO: Reference the heap sort section.

TODO: Reference the section on standard implementations.

1.8.8 References

- **Wikipedia: Heap (data structure)**: The worded definition was taken from here.
- **Wikipedia: Binary Heap**: Algorithms were taken from here.

1.9 Tries

TODO: Write this section!

1.10 Hash Tables

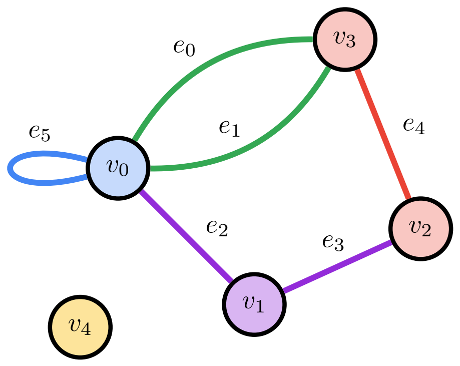
TODO: Write this section!

Chapter 2

Graphs

2.1 Basic Definitions

A graph G consists of two finite sets: a nonempty set $V(G)$ of *vertices*, and a set $E(G)$ of *edges*, where each edge is associated with a set consisting of either one or two vertices called its *endpoints*.



$\{v_0, v_1, v_2, v_3, v_4\}$ are *vertices*.

$\{e_0, e_1, e_2, e_3, e_4, e_5\}$ are *edges*.

Edge e_4 is *incident* on each *endpoint* v_2 and v_3.

Vertices v_2 and v_3 are *adjacent* (connected by an edge).

Edges *connect* their endpoints.

Edges e_2 and e_3 are *adjacent* (shared endpoint).

Edges e_0 and e_1 are *parallel* (same pair of endpoints).

Edge e_5 is a *loop* (only one endpoint).

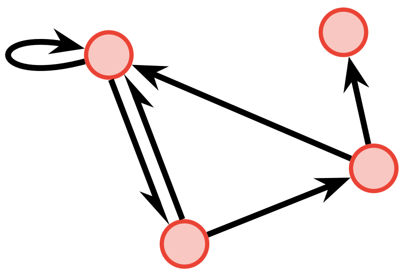
Vertex v_0 is a *adjacent to itself* (endpoint of a loop).

Vertex v_4 is *isolated* (no incident edges).

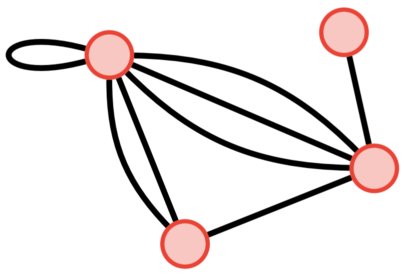
Figure 2.1: Basic parts of a *graph*.

Instead of $E(G)$, a *directed graph* (or *digraph*) has a set $D(G)$ of *directed edges*. A *mixed graph* can contain both directed and undirected edges.

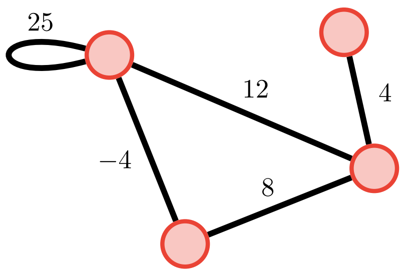
A *weighted graph* has valued edges. A *multigraph* may have parallel edges. A *simple graph* has no loops or parallel edges.



(a) Directed graph.



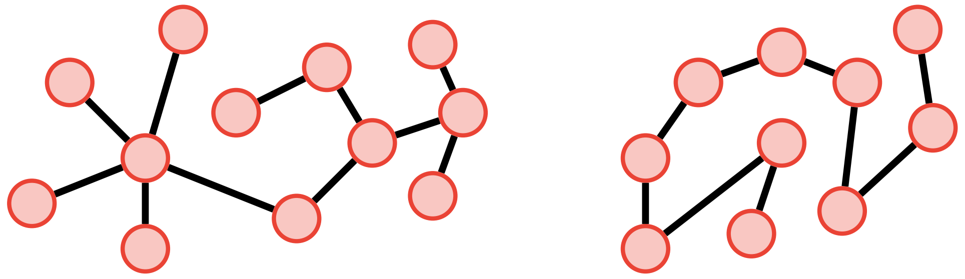
(b) Undirected multigraph.



(c) Weighted undirected graph.

Figure 2.2

A *list* is a graph with only one *path*. A *tree* is a graph with exactly one path between each pair of vertices.



(a) A tree.

(b) A list.

Figure 2.3

(Many more graph variations exist, but an exhaustive list is beyond the scope of a quick reviewer.)

A *walk* is an alternating sequence of adjacent vertices and edges. A walk with only one vertex (and no edges) is a *trivial walk*. A *closed* walk starts and ends on the same vertex.

A *trail* is a walk with no repeated edges. A *circuit* is a closed trail with at least one edge.

A *path* is a walk with no repeated edges or vertices. A *simple circuit* is a closed path except it starts and ends on the same vertex, and has at least one edge.

TODO: Diagram for path? Also, we should rework these walk/trail/path and related definitions.

TODO: Define* complete graphs K_n*. Make diagrams for them. (Epp, Page 633)

TODO: Define* complete bipartite graphs $K_{m,n}$*. Make diagrams for them. (Epp, Page 633)

TODO: Define* subgraph*. (Epp, Page 634)

Two vertices are *connected* iff there is walk between them. A graph is *connected* iff there is a walk between every pair of its vertices.

A graph H is a *connected component* of its supergraph G iff H is connected, and no connected subgraph of G is a supergraph of H that also contains vertices and edges not in H. ***TODO: Can we simplify this definition? Also, we should draw diagrams for this.***

The *degree* of a vertex (denoted $\deg(v)$ for the degree of vertex v) is the number of edges incident on it, with loops counted twice. The *total degree* of a graph is the sum of the degrees of all vertices in the graph.

2.2 Graph Representation

TODO: this

2.3 More Theoretical Results

TODO: What theoretical results are useful to include here? Should we create a separate section for stuff towards the end of the chapter that aren't quite as useful for typical algorithms problems?

TODO: Include **degree sum formula*****: The sum of all degrees of a graph is twice the number of its edges. Corollary: the total degree of the graph is also be even. (Epp, Page 636)***

TODO: Include **handshake lemma** ***(lemma of degree sum formula): There is an even number of vertices of odd degree.***

TODO: Consider including connectedness lemmas. (Epp, Page 647)

TODO: Königsberg bridge problem: If a graph has an Euler circuit, every vertex has positive even degree. If a connected graph has positive even degree, it has an Euler circuit. If a graph has odd degree, then it has no Euler circuit. A connected graph has an Euler circuit iff every vertex has positive degree. (Epp, Page 652)

TODO: The corollary on Epp Page 653: There is an Euler path iff the graph is connected, the start and end vertices have odd degree, and all other vertices have even degrees.

2.4 More Definitions

TODO: This is actually a temporary section and will probably not be in the final version. I'm just using it to write down some interesting definitions that might be useful later.

TODO: Consider adding **Euler circuit*****: A circuit that contains every vertex and edge of a graph. (That is, every edge is used exactly once, but vertices can be used multiple times.)***

TODO: Consider adding **Euler trail*****: A trail that passes through every edge of a graph once, and every vertex at least once.***

2.5 References

- *Discrete Mathematics with Applications* (4th edition) by Susanna S. Epp, Chapter 10
  - A major resource I used to learn about graphs and write this section out. Highly recommended to go through it yourself if you're still learning.
  - I like the first sentence of the definition on page 626, so I copied it verbatim for the first sentence of Section 2.1.
- **Wikipedia**

Chapter 3

Divide-and-Conquer Algorithms

3.1 Introduction

TODO: What is divide-and-conquer?

TODO: Motivate why the examples provided are interesting to look at.

TODO: What is the master algorithm for divide-and-conquer?

3.2 Karatsuba Multiplication

TODO: this

3.3 Fast Fourier Transform (FFT)

TODO: this

Chapter 4

Dynamic Programming

4.1 Introduction

TODO: What is dynamic programming?

TODO: Motivate why the examples provided are interesting to look at.

4.2 Fibonacci Sequence

4.2.1 Introduction

The *Fibonacci sequence* is a sequence in which each number is the sum of the two preceding ones. The sequence commonly begins with 0 and 1, so the sequence is written as:

$$0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, 144, \ldots$$

Mathematically, the Fibonacci sequence is commonly defined by the following recurrence relation:

$$F_0 = 0 \qquad F_1 = 1 \qquad\qquad F_n = F_{n-1} + F_{n-2} \quad \text{for } n > 1$$

For our purposes, we are interested in computing a single term of the sequence.

4.2.2 Naive Recursive Algorithm

A very simple algorithm is to simply implement our definition as a recursive function, though the runtime performance is a nasty $O(2^n)$. Calculating F_{42} took 45 seconds with a AMD Ryzen 7 5800HS CPU:

Listing 4.1: Python code for a naive recursion algorithm. Time: $O(2^n)$.

```
def fib(n):
    if n <= 1: return n
    return fib(n - 1) + fib(n - 2)

print(fib(12)) # 144

from timeit import timeit
timeit(lambda : fib(42), number=1) # approx. 45 seconds
```

4.2.3 Top-Down Memoization Algorithm

One way to speed it up is the recognition that once we've computed a term of the sequence, we can store it for future use. This algorithm runs in $O(n)$ time since we calculate each term only once:

Listing 4.2: Python code for a memoized recursion algorithm.

```
def fib(n):
    memo = {0: 0, 1: 1}
    def _fib(n):
        if n not in memo: memo[n] = _fib(n - 1) + _fib(n - 2)
        return memo[n]
    return _fib(n)

print(fib(12)) # 144

from timeit import timeit
timeit(lambda : fib(42), number=1) # approx. 1.8e-05 seconds (18 microseconds)
timeit(lambda : fib(994), number=1) # approx. 0.00040 seconds (400 microseconds)
timeit(lambda : fib(995), number=1) # Python 3.10.9 'maximum recursion depth exceeded' error
```

Interestingly, if we calculate for $n = 995$ or higher, Python throws a `maximum recursion depth exceeded` *error.*

4.2.4 Bottom-Up Algorithm

You may have already spotted that rather than start from F_n and recursively figuring out what we need to solve it, we could instead start from F_2 (and hardcode the base cases F_0 and F_1) and work our way upwards. This algorithm also runs in $O(n)$:

Listing 4.3: Python code for a bottom-up algorithm.

```
def fib(n):
    if n <= 1: return n
    a, b = 0, 1
    for _ in range(1, n):
        a, b = b, a + b
    return b

print(fib(12)) # 144

from timeit import timeit
timeit(lambda : fib(42), number=1) # approx. 4.7e-06 seconds (4.7 microseconds)
timeit(lambda : fib(994), number=1) # approx. 3.9e-05 seconds (39 microseconds)
timeit(lambda : fib(1000000), number=1) # approx. 5.6 seconds
```

Usefully, we also

4.2.5 References

- **Wikipedia**: The mathematical notation was taken from here. Also, I stole a bit of the wording from it for the introduction.

4.3 0-1 Knapsack Problem

4.3.0.1 Problem Statement

You have a bag with a maximum weight carrying limit, and a set of items. Each item has a weight and dollar value. Find a subset of items with the largest dollar value possible that can fit within in the bag's weight limit.

There is only one copy of each item available for you to take. Each item is either taken in its entirety, or not taken at all. You cannot "partially take" an item.

4.3.1 Example

Suppose your bag can only carry up to 15 kg in total. The possible items we can choose from are shown in Table 4.1.

Table 4.1: The set of items we must choose from.

	weight	value
Item 1	12 kg	\$4
Item 2	4 kg	\$10
Item 3	2 kg	\$2
Item 4	1 kg	\$2
Item 5	1 kg	\$1

The optimal solution is to take all items except Item 1. The total value of these items is $\$10 + \$2 + \$2 + \$1 = \$15$. The total weight of these items is $4\,\mathrm{kg} + 2\,\mathrm{kg} + 1\,\mathrm{kg} + 1\,\mathrm{kg} = 8\,\mathrm{kg}$, which is clearly within the bag's 15 kg limit.

If we took Item 1, we will only have 3 kg remaining. We wouldn't be able to take Item 2, which is a significantly valuable item in the list. The best we can do is to take Item 3 and Item 5, giving us a total of $\$4 + \$2 + \$1 = \$7$.

4.3.2 Naive Brute Force Solution

We can try enumerating every possible subset of items:

			Items				
#	1	2	3	4	5	Weight	Value
1						0 kg	\$0
2	Take					12 kg	\$4
3		Take				4 kg	\$10
4	Take	Take				16 kg	\$14
5			Take			2 kg	\$2
6	Take		Take			14 kg	\$6
7		Take	Take			6 kg	\$12
8	Take	Take	Take			18 kg	\$16
9				Take		1 kg	\$2
10	Take			Take		13 kg	\$6
11		Take		Take		5 kg	\$12
12	Take	Take		Take		17 kg	\$16
13			Take	Take		3 kg	\$4
14	Take		Take	Take		15 kg	\$8
15		Take	Take	Take		7 kg	\$14
16	Take	Take	Take	Take		19 kg	\$18
17					Take	1 kg	\$1
18	Take				Take	13 kg	\$5
19		Take			Take	5 kg	\$11
20	Take	Take			Take	17 kg	\$15
21			Take		Take	3 kg	\$3
22	Take		Take		Take	15 kg	\$7
23		Take	Take		Take	7 kg	\$13
24	Take	Take	Take		Take	19 kg	\$17
25				Take	Take	2 kg	\$3
26	Take			Take	Take	14 kg	\$7
27		Take		Take	Take	6 kg	\$13
28	Take	Take		Take	Take	18 kg	\$17
29			Take	Take	Take	4 kg	\$5
30	Take		Take	Take	Take	16 kg	\$9
31		Take	Take	Take	Take	8 kg	\$15
32	Take	Take	Take	Take	Take	20 kg	\$19

By rejecting all subsets above the weight limit (15 kg) and getting the highest-value subset, we can clearly see that this will lead to an optimal solution. However, the runtime complexity is $O(2^n)$.

4.3.3 Bottom-Up Tabulation Algorithm

Let:

- W = the bag's carrying weight limit.
- N = the number of items we can choose from.
- $\{S_1, S_2, \ldots, S_I\}$ = the set of all items we can choose from. *(This intentionally uses one-based indexing.)*

- w_i = the weight of item S_i.
- v_i = the dollar value of item S_i.

For this solution, we will assume integer weight values, $W \geq 0$, and $w_i \leq W$ for all possible w_i.

Our main data structure will be a table $m[i, w]$ with size $(N + 1) \times (W + 1)$. For example:

Table 4.2: An initial table $m[i, w]$ for **?? ??**.

		w															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1																
	2																
	3																
	4																
	5																

i corresponds to the subset of items we have currently selected from. For **?? ??**:

- $i = 0$ means we have selected from an empty set $\{\}$.
- $i = 1$ means we have selected from $\{S_1\}$.
- $i = 2$ means we have selected from $\{S_1, S_2\}$.
- $i = 3$ means we have selected from $\{S_1, S_2, S_3\}$.
- $i = 4$ means we have selected from $\{S_1, S_2, S_3, S_4\}$.
- $i = 5$ means we have selected from $\{S_1, S_2, S_3, S_4, S_5\}$.

w corresponds to hypothetical weight limits up until the weight limit W. For example:

- $w = 0$ corresponds to a bag of weight limit 0 kg,
- $w = 1$ corresponds to a bag of weight limit 1 kg,
- $w = 2$ corresponds to a bag of weight limit 2 kg, etc.

Each cell of $m[i, w]$ is the highest total dollar value possible for the given i and w.

To fill the remaining cells, we must apply an optimal substructure. This can be expressed simply as:

$$m[0, w] = 0 \tag{4.1}$$
$$m[i, w] = \max(m[i-1, w-w_i] + v_i,\ m[i-1, w]) \tag{4.2}$$

(4.1) states the trivial case where choosing from zero items will always optimally total zero value. This lets us prefill zeroes in the first row as shown in Table 4.2.

(4.2) is a recursive equation based around a yes/no decision on whether or not to include an item in the bag. Importantly, this equation only references the previous row $i - 1$, meaning that we must calculate each row in order starting from $i = 1$. Without concerning ourselves with the details of the calculation, let's fill out this first row $i = 1$:

		w															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
	2																
	3																
	4																
	5																

Using the results of row $i = 1$, we can now calculate row $i = 2$:

		w															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
	2	0	0	0	0	10	10	10	10	10	10	10	10	10	10	10	10
	3																
	4																
	5																

...and so on.

To understand how equation (4.2) works and how we've been calculating these rows, let's use the $i = 3$ row as an example since this is where things get more interesting:

		w															
		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
	2	0	0	0	0	10	10	10	10	10	10	10	10	10	10	10	10
	3	0	0	2	2	10	10	12	12	12	12	12	12	12	12	12	12
	4																
	5																

Let's take a closer look at how $m[3,5]$ is calculated. $m[3,5]$ is the highest dollar value of a bag with a carrying capacity of 5 kg, selecting only from the set of items $\{S_1, S_2, S_3\}$. If we apply equation (4.2), we get:

$$w_3 = 2\,\text{kg} \qquad v_3 = \$2$$

$$\begin{aligned}
m[3,5] &= \max(m[i-1, w-w_3] + v_3,\ m[i-1, w]) \\
&= \max(m[3-1, 5-w_3] + v_3,\ m[3-1, 5]) \\
&= \max(m[2, 5-w_3] + v_3,\ m[2, 5]) \\
&= \max(m[2, 5-2] + 2,\ m[2, 5]) \\
&= \max\big(\underbrace{m[2,3] + 2}_{\text{Option 1}},\ \underbrace{m[2,5]}_{\text{Option 2}}\big) \qquad (4.3)
\end{aligned}$$

The last line (4.3) reads that we choose the best dollar value of two options:

Option 1: *We put item S_3 in the bag.* To calculate this option, we are essentially taking the empty 5 kg bag, putting item S_3 into it, and then asking for the best dollar value that we can put in the space that remains in the bag using only the previous items $\{S_1, S_2\}$. The weight of item S_3 is $w_3 = 2\,\text{kg}$, so the remaining space is $w - w_3 = 5 - 2 = 3\,\text{kg}$.

Option 2: *We don't put item S_3 in the bag.* To calculate this option, we ask what the highest dollar value of a bag with a carrying capacity of the same 5 kg, except we select only from items $\{S_1, S_2\}$.

Continuing on from (4.3):

$$\begin{aligned} m[3,5] &= \max(0+2,\ 10) \\ &= \$10 \end{aligned}$$

This matches what is written in the cell.

Continuing until the whole table is complete, we get:

		w 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
	1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
	2	0	0	0	0	10	10	10	10	10	10	10	10	10	10	10	10
	3	0	0	2	2	10	10	12	12	12	12	12	12	12	12	12	12
	4	0	2	2	4	10	12	12	14	14	14	14	14	14	14	14	14
	5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

Thus, the best dollar value that we can put in a 15 kg bag using items $\{S_1, S_2, S_3, S_4, S_5\}$ is $m[5,15] = \$15$.

To read back an optimal set of items, we must *backtrack* starting from $m[5,15]$. The idea is to first look directly backwards to see if the weight is the same, otherwise we subtract the value and the weight and check the appropriate cell. If done correctly, we touch the following cells:

Table 4.3: Backtracking through the table for **??** **??**.

						w 0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				i	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S_1	$\rightarrow$	12 kg	\$4		1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
S_2	$\rightarrow$	4 kg	\$10		2	0	0	0	0	10	10	10	10	10	10	10	10	10	10	10	10
S_3	$\rightarrow$	2 kg	\$2		3	0	0	2	2	10	10	12	12	12	12	12	12	12	12	12	12
S_4	$\rightarrow$	1 kg	\$2		4	0	2	2	4	10	12	12	14	14	14	14	14	14	14	14	14
S_5	$\rightarrow$	1 kg	\$1		5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

To understand how it works, let's start from the beginning at $m[5,15]$:

					w															
					0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
				i 0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
S_1	$\to$	12 kg	\$4	1	0	0	0	0	0	0	0	0	0	0	0	0	4	4	4	4
S_2	$\to$	4 kg	\$10	2	0	0	0	0	10	10	10	10	10	10	10	10	10	10	10	10
S_3	$\to$	2 kg	\$2	3	0	0	2	2	10	10	12	12	12	12	12	12	12	12	12	12
S_4	$\to$	1 kg	\$2	4	0	2	2	4	10	12	12	14	14	14	14	14	14	14	14	14
S_5	$\to$	1 kg	\$1	5	0	2	3	4	10	12	13	14	15	15	15	15	15	15	15	15

Directly behind $m[5,15]$ is $m[4,15]$. Here, we check for one of two possible branches

Branch 1: If $m[5,15] \neq m[4,15]$, then we know that *our optimal set does contain item* S_5. This is because the optimal set's composition <u>must have changed</u> when we introduced the new item S_5.

Branch 2: If $m[5,15] = m[4,15]$, then we say that *our optimal set does not necessarily contain item* S_5. This is because the optimal set's composition <u>doesn't need to have changed</u> when we introduced the new item S_5.

Side note: Branch 2 is carefully worded to say that it's possible but uncertain that item S_5 can be included in the optimal set, but that we know for certain that it can be left out, hence we "say" that the optimal set does not necessarily contain S_5. It simplifies our discussion to skip over checking if we can include S_5.

In this example, we see that $m[5,15] \neq m[4,15]$, so we take item S_5 as part of the optimal set.

Now, depending on the branch, we will visit a new cell. This is simply the corresponding cell as described by equation (4.2):

If we took Branch 1: We visit $m[i-1, w-w_i]$. In this case, we visit $m[4,14]$.

If we took Branch 2: We visit $m[i-1, w]$. In this case, we visit $m[4,15]$.

This process can repeat until we arrive at the $i = 0$ row.

Reading off of Table 4.3, we:

- took Branch 1 at $m[5,15]$,
- took Branch 1 at $m[4,14]$,
- took Branch 1 at $m[3,13]$,
- took Branch 1 at $m[2,11]$, and
- took Branch 2 at $m[1,7]$.

Therefore, our optimal set is $\{S_5, S_4, S_3, S_2\}$, which has a total value of \$15.

This tabulation algorithm runs in $O(NW)$ time and space. Further minor improvements to the algorithm are possible and is left as an exercise for the reader.

4.3.4 Naive Top-Down Recursion Algorithm

TODO: this?

4.3.5 Top-Down Memoization Algorithm

TODO: this?

4.3.6 References

- **Wikipedia**: The example and the mathematical notation was taken from here.
- **GeeksforGeeks**: Online practice problem.

Chapter 5

Randomized Algorithms

5.1 Introduction

TODO: Start this!

Chapter 6

Mathematical Computing

6.1 Introduction

TODO: There should be a better name for this chapter since the scope is a little wider than the name suggests. Some of this stuff should maybe be moved to the intro chapter though.

TODO: How to compute square root? Linear search vs. binary search vs. Newton's method? What about fast inverse square root?

TODO: Machine epsilon

TODO: How to manage overflow and underflow?

Chapter 7

Standard Implementations

7.1 Javascript

TODO: Compile useful built-ins!

7.1.1 Max-Heap Implementation

TODO: Explain this implementation? Why is it only useful for leetcode/hackerrank? What limitations does it have?

TODO: Better formatting?

Listing 7.1: A simple Javascript *max-heap* implementation.

```
function heapify(heap) {
    for (let i = (heap.length >> 1) - 1; i >= 0; --i)
    _downHeapify(heap, i);
}

function hpush(heap, item) {
    let curr = heap.length;
    heap.push(item);
    _upHeapify(heap, curr);
}

// Assumes heap.length > 0
function hpop(heap) {
    if (heap.length === 1) return heap.pop();
    const toReturn = heap[0];
    heap[0] = heap.pop();
    _downHeapify(heap, 0);
    return toReturn;
}

// Assumes heap.length > 0
function hpushpop(heap, item) {
    if (item[0] < heap[0][0]) {
        [item, heap[0]] = [heap[0], item];
        _downHeapify(heap, 0);
    }
    return item;
}

function hpopManyWithoutKey(heap, n) {
    const toReturn = new Array(n);
    for (let i = 1; i < n; ++i) {
        toReturn[i] = heap[0][1];
        heap[0] = heap.pop();
        _downHeapify(heap, 0);
    }
    toReturn[0] = heap[0][1];
    return toReturn;
}
```

```
function hpopManyWithoutReturn(heap, n) {
    if (heap.length === n) {
        heap.splice(0, heap.length);
    } else {
        for (let i = 1; i < n; ++i) {
            heap[0] = heap.pop();
            _downHeapify(heap, 0);
        }
        heap[0] = heap.pop();
    }
}

function _downHeapify(heap, i) {
    const left = (i << 1) + 1;
    if (heap[left] === undefined) return;
    const right = left + 1;
    const next = (heap[right] === undefined || heap[
    left][0] > heap[right][0]) ? left : right;
    if (heap[next][0] <= heap[i][0]) return;
    [heap[next], heap[i]] = [heap[i], heap[next]];
    _downHeapify(heap, next);
}
function _upHeapify(heap, i) {
    if (i === 0) return;
    const parent = (i - 1) >> 1;
    if (heap[parent][0] > heap[i][0]) return;
    [heap[parent], heap[i]] = [heap[i], heap[parent]];
    _upHeapify(heap, parent);
}
```

7.2 Python

TODO: Compile useful stuff from the standard library here! Examples:

- `functools` (including `cache`, `lru_cache`, etc.)
- `heapq`
- `bisect`

Appendix A

Appendices

A.1 Introduction to Fourier Analysis, for Computer Science Students

A.1.1 Background

TODO: this