

# HT2026 Design & Analysis of Algorithms notes

Remaining **TODOs**: 34

Relevant reading:

T. H. Cormen, C. E. Leiserson, R. L. Rivest and C. Stein. *Introduction to Algorithms*

## Contents

1. Program cost and asymptotic notation .....	2
2. Divide and conquer algorithms .....	4
3. Data structures: heaps and queues .....	10
4. Dynamic programming .....	13
5. Graph decomposition .....	15
6. Shortest paths in graphs .....	21
7. Greedy algorithms .....	26
8. Stable matching .....	30
Index .....	33

## 1. Program cost and asymptotic notation

**Definition 1.1:** An **algorithm** is a finite set of well-defined instructions to accomplish a specific task.

**Definition 1.2:** An **efficient** algorithm runs in polynomial time.

**Definition 1.3:** **Insertion sort** compares each  $(i + 1)$ th element and compares it with the previously sorted  $i$  elements, inserting it in the correct place.

In CLRS-style pseudocode (as used in *Introduction to Algorithms*):

Input: An 1-indexed array A of integers

Output: Array A is sorted in non-decreasing order

InsertionSort(A):

```

for j = 2 to A.length
  key = A[j + 1]
  // insert A[j + 1] into the sorted sequence A[1..j]
  i = j
  while i > 0 and A[i] > key
    A[i + 1] = A[i]
    i = i - 1
  A[i + 1] = key

```

**Definition 1.4:** The running time of a CLRS program is defined as:

- Line  $i$  takes constant time  $c_i$
- When a loop exits normally, the test is executed one more time than the loop body

**Remark:** The running time of insertion sort as given is  $T(n) = c_1n + c_2(n - 1) + c_3(n - 1) + c_4(n - 1) + c_5 \sum_{j=1}^{n-1} t_j + c_6 \sum_{j=1}^{n-1} (t_j - 1) + c_7 \sum_{j=1}^{n-1} (t_j - 1) + c_8(n - 1)$  where  $t_j$  is the number of times the test of the while loop is executed for a given value of  $j$ .

Then in the worst case,  $t_j = j + 1$ , so  $T(n) = an^2 + bn + c$  for some  $a, b, c$ . Hence  $T(n)$  is quadratic in  $n$ .

In the best case,  $t_j = 1$  so  $T(n)$  is linear.

**Definition 1.5:** Let  $f, g : \mathbb{N} \rightarrow \mathbb{R}^+$ . Then

$$O(g(n)) := \{f : \mathbb{N} \rightarrow \mathbb{R}^+ \mid \exists n_0 \in \mathbb{N}^+. \exists c \in \mathbb{R}^+. \forall n. n \geq n_0 \implies f(n) \leq c \cdot g(n)\}.$$

If  $f \in O(g(n))$  then  $g$  is an **asymptotic upper bound** for  $f$ .

**Proposition 1.6:** The algorithm is correct.

*Proof:*

By a **loop-invariant** argument:

- **Initialisation** - Prove the invariant  $I$  holds prior to first iteration
- **Maintenance** - Prove that if  $I$  holds just before an iteration, then it holds just before the next iteration
- **Termination:** Prove that, when the loop terminates, the invariant  $I$  along with the reason the loop terminates imply the correctness of the program

This is similar to mathematical induction, but rather than proving for all numbers, we expect to exit the loop.

Invariant: At the start of the  $j$ th iteration,  $A[1..j]$  is sorted.

When  $j = 1$ ,  $A[1..j]$  is a singleton so is trivially sorted.

The outer loop terminates when  $j = A.length$ . So the loop invariant at termination says that  $A[1..A.length] = A$  is sorted.

To prove maintenance, we need to prove that, at the end of the while loop, the sequence  $A[1], \dots, A[i], key, A[i+2], \dots, A[j+1]$  are sorted.

The invariant of the while loop is: **TODO**

□

**Remark:** Some nice properties of insertion sort:

- It is stable (preserves relative order of equal keys)
- In-place
- Online (can sort the list as it is recieved)

**Lemma 1.7:** Let  $f, g, h : \mathbb{N} \rightarrow \mathbb{R}^+$ . Then:

- $\forall c > 0 . f \in O(g) \implies cf \in O(g)$
- $\forall c > 0 . f \in O(g) \implies f \in O(cg)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \implies f_1 + f_2 \in O(g_1 + g_2)$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \implies f_1 + f_2 \in O(\max(g_1, g_2))$
- $f_1 \in O(g_1) \wedge f_2 \in O(g_2) \implies f_1 \cdot f_2 \in O(g_1 \cdot g_2)$
- $f \in O(g) \wedge g \in O(h) \implies f \in O(h)$
- $\forall l > 0 . \forall p \in \mathcal{P}_l . p(n) \in O(n^l)$  where  $\mathcal{P}_l$  is the set of  $l$ -degree polynomials
- $\forall c > 0 . \lg(n^c) \in O(\lg(n))$
- $\forall c, d > 0 . \lg^c(n) \in O(n^d)$
- $\forall c > 0, d > 1 . n^c \in O(d^n)$
- $\forall 0 \leq c \leq d . c^n \in O(d^n)$

**Definition 1.8:** If  $f(n) \in \Omega(g(n))$ , we say that  $g$  is an **asymptotic lower bound** for  $f$ .

**Corollary 1.9:**  $f(n) \in O(g(n)) \iff g(n) \in \Omega(f(n))$ .

**Definition 1.10:** If  $f(n) \in \Theta(g(n))$ ,  $g$  is an **asymptotic tight bound** for  $f$ .

$$f(n) \in O(g(n)) \wedge f(n) \in \Omega(g(n)) \iff f(n) \in \Theta(g(n)).$$

**Remark:** Big  $O/\Omega/\Theta$  are not closed under function composition.

## 2. Divide and conquer algorithms

**Definition 2.1:** A **divide and conquer algorithm** divides the problem into subproblems, solves each subproblem separately and combines the results.

**Remark:** In general, a divide and conquer algorithm working on an input of size  $n$  can be defined as:

- If  $n$  is small,  $n \leq \ell$  for some constant  $\ell$ , use constant-time brute force solution
- Otherwise, divide the problem into  $a$  subproblems, each  $\frac{1}{b}$  the size of the original
- Let the time to divide a size- $n$  problem be  $D(n)$
- Let the time to combine solutions be  $C(n)$ .

Let  $D(n)$  be the time taken to split a size- $n$  problem up, and  $C(n)$  be the time taken to combine solutions of subproblems.

Then the running time,  $T(n)$ , of a divide-and-conquer algorithm can in general be expressed by the following recurrence:

$$T(n) = \begin{cases} c & \text{if } n \leq \ell \\ aT\left(\frac{n}{b}\right) + D(n) + C(n) & \text{if } n > \ell. \end{cases}$$

**Theorem 2.2** (Master Theorem):

Suppose

$$T(n) \leq aT\left(\left\lceil \frac{n}{b} \right\rceil\right) + O(n^d).$$

Then

$$T(n) = \begin{cases} O(n^d) & \text{if } d > \log_b a \\ O(n^d \log_b n) & \text{if } d = \log_b a \\ O(n^{\log_b a}) & \text{if } d < \log_b a. \end{cases}$$

*Proof:*

TODO

□

*Example (merge sort):*

1. Split array into 2 ( $O(1)$ )
2. Carry out merge sort of on the subarrays (or for singletons, do nothing)
3. Merge the elements from the sorted sublists in order, which is easy as we just pick the smallest first element from each

Pseudocode:

```
MergeSort(A, p, r):
  if r > p + 1
    q = ⌊(p + r) / 2⌋
    MergeSort(A, p, q)
    MergeSort(A, q + 1, r)
    Merge(A, p, q, r)
```

Where Merge is defined as:

```
// Merge A[p..q] and A[p+1..r] so that A[p..r] is sorted
Merge(A, p, q, r):
  n1 = q - p + 1
  n2 = r - q
  let L[1..n1 + 1] and R[1..n2 + 1] be new arrays
  for i = 1 to n1:
    L[i] = A[p + i - 1]
  for j = 1 to n2:
    R[j] = A[q + j]
  // Set the last values of the arrays to be larger than
  // anything else we could encounter, so we don't need
  // to check if we have exhausted all the elements of
  // either array if they aren't balanced
  L[n1 + 1] = ∞
  L[n2 + 1] = ∞
  i = 1
  j = 1
  for k = p to r:
    if L[i] ≤ R[j]:
      A[k] = L[i]
      i = i + 1
    else:
      A[k] = R[j]
      j = j + 1
```

**Remark:** Merging should be left-biased (i.e. in case of tie, pick the left one) so that merge sort is a stable sort.

Merge sort is not in-place (requires  $\Theta(n)$  extra space) and is not online.

Merge is clearly linear.

Then for MergeSort, in the form given above we have  $D(n) = \Theta(1)$ ,  $a = b = 2$  and  $C(n) = \Theta(n)$ .

Then

$$T(n) = \begin{cases} c & \text{if } n = 1 \\ 2T(\frac{n}{2}) + \Theta(n) & \text{if } n > 1 \end{cases}$$

for some constant  $c$ .

There are a few approaches we could use to solve for  $T(n)$ :

- Guess a solution, and use induction to find constants and prove that it works
- Draw out a recursion tree and sum each level, either to obtain an exact answer or to gain a heuristic to guess and check
- Or use the Master Theorem.

Using the Master Theorem, we get that  $T(n) = n \log n$ .

*Example (Integer multiplication):* Suppose we want to multiply two  $n$ -bit integers,  $x$  and  $y$ .

A simple divide-and-conquer approach would be to split each number into a top and bottom half,  $x_l, x_r, y_l, y_r$ , then

$$\begin{aligned} xy &= (2^{n/2}x_l + x_r)(2^{n/2}y_l + y_r) \\ &= 2^n x_l y_l + 2^{n/2}(x_l y_r + y_l x_r) + x_r y_r. \end{aligned}$$

Therefore we can compute four multiplications and combine them with three additions (which are linear time in  $n$ ).

The time complexity of this is  $T(n) = 4T(\frac{n}{2}) + O(n)$ , which by the Master Theorem is  $O(n^2)$ .

However, we can do better.

**Definition 2.3:** The **Karatsuba algorithm**, or **Karatsuba-Ofman algorithm**, is a subquadratic algorithm for integer multiplication. It utilises the fact that

$$x_l y_r + y_l x_r = (x_l + y_l)(x_r + y_r) - x_l x_r - y_l y_r,$$

so we only need three multiplications. This needs more additions, but that is still linear time.

The recurrence is now  $T(n) = 3T(\frac{n}{2}) + O(n) = O(n^{\log_3 2}) \approx O(n^{1.59})$ .

*Example (matrix multiplication):*

A Naïve matrix multiplication algorithm, based on the definition, operating on  $n \times n$  matrices has time complexity  $O(n^3)$ , as it needs to compute  $n$  multiplications,  $n^2$  times.

A naïve divide-and-conquer approach might split each matrix into four  $\frac{n}{2} \times \frac{n}{2}$  matrices, and compute  $\mathbf{C} = \mathbf{AB}$  as

$$\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{A}_{11} & \mathbf{A}_{12} \\ \mathbf{A}_{21} & \mathbf{A}_{22} \end{pmatrix} \begin{pmatrix} \mathbf{B}_{11} & \mathbf{B}_{12} \\ \mathbf{B}_{21} & \mathbf{B}_{22} \end{pmatrix}.$$

This requires computing 8 subproblems ( $\mathbf{A}_{11}\mathbf{B}_{11}, \mathbf{A}_{12}\mathbf{B}_{11}$  etc), and the addition of the submatrices is quadratic, so the time complexity is

$$\begin{aligned} T(n) &= 8T\left(\frac{n}{2}\right) + O(n^2) \\ &= O(n^3) \text{ by the Master Theorem.} \end{aligned}$$

This is no better than the previous approach, but it is possible to improve upon this.

**Definition 2.4:** **Strassen's algorithm** utilises a divide-and-conquer approach with only 7 subproblems.

$$\begin{pmatrix} \mathbf{C}_{11} & \mathbf{C}_{12} \\ \mathbf{C}_{21} & \mathbf{C}_{22} \end{pmatrix} = \begin{pmatrix} \mathbf{P}_4 + \mathbf{P}_5 - \mathbf{P}_2 + \mathbf{P}_6 & \mathbf{P}_1 + \mathbf{P}_2 \\ \mathbf{P}_3 + \mathbf{P}_4 & \mathbf{P}_1 + \mathbf{P}_5 - \mathbf{P}_3 - \mathbf{P}_7 \end{pmatrix},$$

where

$$\begin{aligned} \mathbf{P}_1 &= \mathbf{A}_{11}(\mathbf{B}_{12} - \mathbf{B}_{22}) \\ \mathbf{P}_2 &= (\mathbf{A}_{11} + \mathbf{A}_{12})\mathbf{B}_{22} \\ \mathbf{P}_3 &= (\mathbf{A}_{21} + \mathbf{A}_{22})\mathbf{B}_{11} \\ \mathbf{P}_4 &= \mathbf{A}_{22}(\mathbf{B}_{21} - \mathbf{B}_{11}) \\ \mathbf{P}_5 &= (\mathbf{A}_{11} + \mathbf{A}_{22})(\mathbf{B}_{11} + \mathbf{B}_{22}) \\ \mathbf{P}_6 &= (\mathbf{A}_{12} - \mathbf{A}_{22})(\mathbf{B}_{21} + \mathbf{B}_{22}) \\ \mathbf{P}_7 &= (\mathbf{A}_{11} - \mathbf{A}_{21})(\mathbf{B}_{11} + \mathbf{B}_{12}). \end{aligned}$$

The recurrence is now  $T(n) = 7T(\frac{n}{2}) + O(n^2)$ , which by the Master Theorem is  $O(n^{\lg 7}) \approx O(n^{2.81})$ .

**Remark:** Suppose a recursion splits into two subproblem with size  $\sqrt{n}$ , and suppose that the division is  $O(1)$  and the combining is  $O(\log n)$ . Then we have  $T(n) = 2T(n^{\frac{1}{2}}) + \log n$ ,

which isn't directly expressible with the Master Theorem. We can substitute  $k = \log n$  to get  $T(2^k) = 2T(2^{\frac{k}{2}}) + k$ ; if we let  $S(k) = T(2^k)$  then  $S(k) = 2S(\frac{k}{2}) + k$ . We can then use the Master Theorem, with  $a = 2, b = 2, d = 1$ , giving that  $S(k) = O(k \log k)$ . Hence  $T(n) = O(\log n \log \log n)$

**Definition 2.5:** **Binary search** searches for the presence or position of an element in a sorted array:

```
// Search for the presence of z in A[p..r]
BinSearch(A, p, r, z):
  if p >= r:
    return "no"
  q = [(p + r) / 2]
  if z == A[q]:
    return "yes"
  else if z < A[q]:
    return BinSearch(A, p, q, z)
  else:
    return BinSearch(A, q + 1, r, x)
```

This has running time  $T(n) \leq T(\lceil \frac{n}{2} \rceil) + O(1)$ ; by the master theorem  $T(n) = O(\log n)$

**Definition 2.6:** The  **$i$ th-order statistic** of a set of  $n$  (distinct) elements is the  $i$ th smallest element - i.e. the element that has exactly  $(i - 1)$  smaller elements.

**Remark:** The median is the  $\lfloor \frac{n+1}{2} \rfloor$ -order statistic.

**Definition 2.7:** The **Select** algorithm finds the  $i$ th-order statistic in linear time. It does this by partitioning the elements of the list around an estimate for the median, based on the medians of sublists of length 5, and from there operating on one of the smaller subgroups.

```
Select(A, i):
  medians = []
  for j = 1 to A.length by 5:
    sort A[i..i+5)
    medians[j / 5] = A[i+2] // "baby median"
  x = Select(medians, [(medians.length + 1) / 2]) // median of medians
  partition A into lt, gt, based on size compared to x
  k = lt.length
  if i = k + 1:
    return x
  else if i <= k:
    return Select(lt, i)
```

```

else if i > k + 1:
    return Select(gt, i - k - 1)

```

How good is the median of medians as an estimate of the true median? The number of elements  $< x$  is at least  $\frac{3n}{10} - 6$ .

*Proof:* **TODO** □

So then the size of each subarray is no more than  $\lfloor 7n/10 + 6 \rfloor$ .

Then it can be proven that the running time is linear, by guessing that  $T(n) = cn$  for some constant  $c$ , and using induction.

*Proof:* **TODO** □

**Theorem 2.8:** The running time of every comparison-based sorting algorithm is  $\Omega(n \log n)$ , i.e. at best it is  $O(n \log n)$ .

*Proof:*

In a decision tree of a comparison-based sorting algorithm on an input of length  $n$ , there are  $n!$  leaves. Every binary tree of depth  $d$  has at most  $2^d$  leaves; so to get to  $n!$  we need a depth of at least  $\log n! = \Omega(n \log n)$ . □

**Definition 2.9:** If we know our elements come from a certain discrete interval, we can use **counting sort**: just count how often each element appears, then insert the relevant number of each elements. This is  $O(n + k)$  ( $= O(n)$  if  $k = O(n)$ ) but is not in-place; however we can design it to be stable. The **height** of a tree is the longest simple path from the root to a leaf; a binary heap with  $n$  nodes has height  $\lfloor \log n \rfloor$ .

Input: An array  $A$  of  $n$  elements with elements in the interval  $[0..k]$

Output: An array  $B$  consisting of a sorted permutation of  $A$

```

CountingSort(A, k):
    create array C of size k + 1
    for i = 0 to k:
        C[i] = 0
    for j = 1 to A.length:
        C[A[j]] += 1
    for i = 1 to k:
        C[i] = C[i] + C[i - 1]
    // C[i] now contains the number of elements <= i
    for j = n downto 1:
        B[C[A[j]]] = A[j]
        C[A[j]] -= 1

```

**Definition 2.10:** The **Fast Fourier Transform (FFT)** can be used to multiply two polynomials of degree  $(n - 1)$  in  $O(n \log n)$  time.

1. Split each polynomial into 2 polynomials of even degree:

$$a_0 + a_1x + a_2x^2 + \dots = (a_0 + a_2x^2 + \dots) + x(a_1 + a_3x^2 + \dots).$$

This is nice because each polynomial is an even function. We now have that  $A(x) = A_{\text{even}(x^2)} + xA_{\text{even}(x^2)}$

2. Represent each polynomial by a list of values at points, which can be computed in  $O(n)$  by Horner's rule,

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \dots))$$

... **TODO**

Essentially, we split into two polynomials and recurse. Once we are in point-value form, we can multiply each point, all in  $O(n)$ , and then convert back to coefficient form (interpolation).

### 3. Data structures: heaps and queues

**Remark:** Unsorted arrays have  $O(1)$  insertion and  $O(n)$  finding the maximum.

**Remark:** Sorted arrays have  $O(n)$  insertion (find position by binary search but then shifting is linear) but  $O(1)$  finding the maximum.

**Corollary 3.1:** It is not possible to have a data structure with  $O(1)$  insertion and  $O(1)$  maximum extraction, as this would give a linear-time comparison-based sorting algorithm which we have seen isn't possible.

**Definition 3.2:** A **heap** is a data structure that is a tree that is completely filled on all levels except the lowest, which is filled from the left to right.

Heaps can be represented with arrays, with each level stored next to each other, read left-to-right and top-to-bottom. This is unambiguous because the tree is complete, so we know how many children each element has.

For a binary tree, if we have the root at  $A[0]$ , the left child of  $A[i]$  is at  $A[2i + 1]$  and the right child is at  $A[2i + 2]$ . If arrays are 1-indexed, we instead get  $A[i]$ 's children at  $A[2i]$  and  $A[2i + 1]$ .

**Definition 3.3:** A **max-heap** is a heap that satisfies the max-heap property.

**Definition 3.4:** The **max-heap property** states that, for every node  $i$  excluding the root, the key of the parent of  $i$  is greater than or equal to the key of  $i$ .

**Remark:** The maximum element of a max-heap is at the root.

**Remark:** Min-heaps can be defined similarly.

**Remark:** Children of a node in a max-heap are not necessarily sorted.

**Definition 3.5:** The **height** of a node in a heap is defined to be the number of edges in the longest simple path from the node to a leaf. The height of a heap is the height of its root; this means that a singleton heap has a height of 0.

**Definition 3.6:** The **MaxHeapify** procedure turns a heap into a max heap, where the two subtrees of the root are already max-heaps. This works by ‘bubbling’ the root down into a permissible location.

Input: A 1-indexed array  $A$  and index  $i$  such that the subtrees of the node  $A[i]$  are max-heaps.

Output: A max heap with root at  $A[i]$ .

```
MaxHeapify(A, i):
  n = A.size
  l = 2i
  r = 2i + 1
  if l <= n and A[l] > A[i]:
    largest = l
  else:
    largest = i
  if r <= n and A[r] > A[largest]:
    largest = r
  if largest != i:
    swap A[i] and A[largest]
    MaxHeapify(A, largest)
```

This is  $O(\log n)$  because the procedure runs at most once for each depth level in the tree; we can also see this by the master theorem: the two subtrees have a maximum size of  $2\frac{n}{3}$ , occurring on the left if the bottom row is exactly half full, so

$$T(n) \leq T\left(\frac{2n}{3}\right) + O(1)$$

$$\implies T(n) = O\left(n^0 \log_{3/2} n\right) = O(\log n).$$

**Definition 3.7:** The **MakeMaxHeap** procedure forms a max heap from an unordered array.

We carry out **MaxHeapify** on every node, starting from the bottom, but skipping leaves, as they are already singleton max-heaps; allows us to skip about half of the nodes.

**MakeMaxHeap(A):**

```
for i = ceil((n + 1) / 2) - 1 downto 1
  MaxHeapify(A, i)
```

This is definitely  $O(n \log n)$ , but in fact a tighter bound is  $O(n)$ .

*Proof:* **MaxHeapify** is linear in the height of the node it operates on; therefore given a node of height  $h$ , **MaxHeapify** takes time  $\leq ch$  for some  $c \in \mathbb{R}^+$ . Moreover, in any binary tree there are at most  $2^{\log n - h}$  nodes at height  $h$ .

So, the running time of **MakeMaxHeap** is

$$\begin{aligned} T(n) &\leq \sum_{h=1}^{\log n} 2^{\log n - h} ch \\ &\leq c \sum_{h=0}^{\infty} 2^{\log n - h} h \\ &= c \sum_{h=0}^{\infty} \frac{n}{2^h} h \\ &= cn \sum_{h=0}^{\infty} \frac{h}{2^h}. \end{aligned}$$

Let  $S := \sum_{h=0}^{\infty} \frac{h}{2^h}$ , and consider  $2S = \sum_{h=0}^{\infty} \frac{h+1}{2^h}$ . Then  $2S - S = \sum_{h=0}^{\infty} \frac{1}{2^h} = 2$ .

Therefore  $T(n) \leq 2cn$ , so  $T(n) = O(n)$ .

□

**Remark:** To extract the maximum element from a max-heap, we extract the root, place the last vertex into the root, and carry out **MakeMaxHeap**. This is  $O(n)$ .

**Remark:** To insert an element, append to the end of the backing array, and carry out **MaxHeapify** recursively **on the parents of the new vertex**. This is  $O(\log n)$ .

**Definition 3.8:** **Heap sort:**

1. Build a max-heap
2. Starting from the root, swap the maximum with the final element in the backing array, and ignore the final node from now on (it remains in the backing array, but we decrement the heap size), then carry out `MaxHeapify` on the root (this is fine because the children are still max-heaps)
3. Repeat until the heap size is 1

This is  $O(n \log n)$  worst case like merge sort, but is in-place like insertion sort.

**Definition 3.9:** A **priority queue** maintains a set of elements, each with an associated key. Max-priority queues give priority to elements with larger keys; min-priority keys are defined similarly.

Max-priority queues have the following operations:

- `Insert(S, x, k)` inserts element  $x$  with key  $k$  into  $S$
- `Maximum(S)` returns the element of  $S$  with the largest key
- `ExtractMax(S)` removes and returns the element of  $S$  with the largest key
- `IncreaseKey(S, x, k)` increases the value of  $x$ 's key to  $k$ , which is assumed to be at least as large as the existing key

If we implement the max-/min-priority queue as a max-/min-heap, both `Insert` and `ExtractMax` are  $O(\log n)$ , and are as described above. `Maximum` is  $O(1)$  and simply returns the root of the heap.

**Definition 3.10:** **IncreaseKey** works by modifying the key, and then bubbles the node up until it satisfies the max-heap property. Note that it does not just call `MaxHeapify` on the parents of the node, as this would unnecessarily visit sibling nodes as well.

Here  $A$  is 1-indexed as before.

```
IncreaseKey(A, i, key):
    require(key >= A[i])
    A[i] = key
    while i > 1 and A[floor(i / 2)] < A[i]:
        swap A[i] and A[floor(i / 2)]
        i = floor(i / 2)
```

This is also  $O(\log n)$  as the while loop cannot run more times than the height of the heap.

## 4. Dynamic programming

**Definition 4.1:** **Dynamic programming** is an optimisation in which we define a sequence of subproblems such that:

- The subproblems are ordered from smallest to largest
- The largest is the one we want to solve
- The optimal solution of a subproblem can be constructed from the optimal solutions of smaller subproblems (this property is **optimal substructure**)

We then solve from smallest to largest and store the solutions.

*Example (Change-making):* Suppose we have lots of coins of different denominations,  $x_1, \dots, x_n$ , and want to give  $u$  units of change.

Then  $C(u) := \min \#$  coins summing up to  $u$ .

More formally,

$$C(u) = \min_{1 \leq i \leq n} \{C(u - x_i) \mid x_i \leq u\} + 1;$$

$$C(0) = 0.$$

The running time is  $O(un)$ . This is polynomial in  $n$  and  $v$  (i.e. in the number of inputs), but is exponential in the size of the inputs.

*Example (Knapsack problem):* A burglar has a knapsack with capacity  $W \in \mathbb{N}$ . There are  $n$  items to pick from, of size/weight  $w_1, \dots, w_n \in \mathbb{N}$  and value  $v_1, \dots, v_n \in \mathbb{N}$ . We want to find the most valuable configuration of items to steal, assuming that there is either 1 of each item, or an unlimited quantity of each.

Assuming first an unlimited supply of each item:

$$K(u) = \max \text{value of items with weight } \leq u$$

$$K(u) := \max_{1 \leq i \leq n} \{K(u - w_i) + v_i \mid w_i \leq u\}$$

$$K(0) = 0.$$

Total running time is  $O(un)$ .

Where there is only one of each item:

$$K(u, S \subseteq \mathbb{N}) = \max \text{value for weight } u \text{ and available items } S;$$

$$K(u, S \subseteq \mathbb{N}) := \max_{i \in S} \{K(u - w_i, S \setminus \{i\}) + v_i \mid w_i \leq u\};$$

$$K(0, S) = K(u, \emptyset) = 0.$$

The running time is  $O(un \cdot 2^n)$ .

But we can modify  $K$  to make it not exponential, by rather than passing a set of indices, passing the largest index that is considered - since if we remove item  $j$ , we should have an optimal solution for  $j - 1$ .

$$K(u, j) = \max\{K(u - w_j, j - 1) + v_j, K(u, j - 1)\}$$

This formulation is stating that for each index  $j$ , we either pick that item or discard it.

Now the running time is  $O(un)$ .

*Example (Longest increasing subsequences):* Task: find the longest increasing subsequences of a sequence; the elements in a subsequence don't need to all be adjacent to each other in the original sequence. **TODO**

**TODO** edit distance

*Example (Longest simple path):* This is an example of a problem that does not have optimal substructure: the longest simple paths from  $A$  to  $B$  and from  $B$  to  $C$  might share a vertex, so cannot be combined. We would therefore need to store not only the length of the longest path of a subproblem, but also the path itself; so dynamic programming cannot be used for this.

However, if the graph is acyclic, the problem does have optimal substructure.

*Example (Travelling salesperson problem):* Given a complete undirected graphs with weights  $d_{ij}$  associated with each edge  $(i, j)$ , find the Hamiltonian cycle with minimal total distance (sum of edge weights).

This is NP-complete, so we don't know if there is a polynomial time solution, but we can verify a solution in polynomial time.

Brute force is very bad, but dynamic programming gives a better (but not polynomial) solution.

Subproblems: for every subset  $S \subseteq \{1, \dots, n\}$  containing 1, and for every  $j \in S, j \neq 1$ , find the shortest path that starts from 1, ends in  $j$ , and passes only once through all the other nodes in  $S$ . Define  $C[S, j]$  to be the length of that path.

Then

$$C[S, j] = \min\{C[S \setminus \{j\}, i] + d_{ij} : i \in S \setminus \{1, j\}\};$$

$$C[\{1\}, \{1\}] = 0.$$

**TODO**

*Example (A better algorithm for longest increasing subsequence):* **TODO**

## 5. Graph decomposition

**Remark:** A directed acyclic graph can be abbreviated to dag.

**Definition 5.1:** An **adjacency matrix** of a graph  $G := (V, E)$  is a  $|V| \times |V|$  matrix  $\mathbf{A}$  where

$$a_{i,j} = \begin{cases} 1 & \text{if } (V_i, V_j) \in E \\ 0 & \text{otherwise.} \end{cases}$$

This has size  $O(|V|^2)$ , but has constant lookup for determining if an edge is present.

For undirected graphs, the adjacency matrix is symmetric.

**Remark:** Given an adjacency matrix  $A$ ,  $(A^n)_{ij}$  is the number of walk of length  $n$  from  $i$  to  $j$ .

If  $A$  represents an undirected graph,  $(A^2)_{i,i}$  is the degree of  $i$ .

**Remark:** An **adjacency list** for a graph  $G := (V, E)$  is a list of  $|V|$  linked lists, one per vertex: the list for vertex  $u$  holds the indices of vertices to which  $u$  has an outgoing edge.

This has size  $O(|V| + |E|)$  but does not allow for checking for the presence of an edge in constant time.

For undirected graphs, if  $u$  is in  $v$ 's adjacency list, then  $v$  is in  $u$ 's.

**Definition 5.2:** **Depth-first search (DFS)** is a linear-time algorithm that tells us what parts of a graph are reachable from a given vertex. It works for both digraphs and undirected graphs.

As soon as a new vertex is discovered, explore from it. As DFS progresses, we assign each vertex a colour:

- not discovered yet (e.g. white)
- discovered, but not fully explored yet (e.g. grey)
- finished (e.g. black)

Input: A graph  $G := (V, E)$

Output: for each vertex  $v \in V$ , a backpointer  $\pi(v)$  (the predecessor of  $v$ ) and two time-stamps, the discovery time  $d[v]$  and finishing time  $f[v]$ .

```
DFS(V, E):
  for u in V:
    colour[u] = white
    pi[u] = null
  time = 0
  for u in V:
    if colour[u] = white:
      DFSVisit(u)
```

```
DFSVisit(u):
  time += 1
  d[u] = time
```

```

colour[u] = grey
for v in neighbours(v):
    if colour[v] = white:
        pi[v] = u
        DFSVisit(v)
time = time + 1
f[u] = time
colour[u] = black

```

Note that  $\forall v \in V . 1 \leq d[v] < f[v] \leq 2|V|$ .

Note that  $\pi, d, f$  are dependent upon the order in which the vertices in the graph are visited, and on the order of the vertices in the adjacency/neighbour lists.

This has running time  $O(|V| + |E|)$  because each vertex is visited once, and  $\text{DFSVisit}(v)$  takes  $\Theta(d(v))$  where  $d(v)$  is the degree of  $v$ ;  $\sum_{v \in V} d(v) = 2|E|$ .

**Definition 5.3:** Let  $E_\pi$  be the edges visited by DFS,

$$E_\pi = \{(\pi[v], v) \mid v \in V, \pi[v] \neq \text{null}\}.$$

Then let  $G_\pi = (V, E_\pi)$  be the **DFS forest**, consisting of **DFS trees** (note that this is a slight abuse of notation because trees and forests are usually undirected).

**Definition 5.4:** A vertex  $u$  is a **descendant** of  $v$  exactly if it is a descendant of  $v$  in the DFS forest, i.e. there is a path from  $v$  to  $u$  in  $G_\pi$ .

**Theorem 5.5** (Parenthesis theorem): For all  $u, v \in V$ , exactly one of the following holds:

1.  $d[u] < f[u] < d[v] < f[v]$  or  $d[v] < f[v] < d[u] < f[u]$  and neither of  $u$  and  $v$  are descendants of each other in the DFS forest
2.  $d[u] < d[v] < f[v] < f[u]$  and  $v$  is a descendant of  $u$  in a DFS tree
3.  $d[v] < d[u] < f[u] < f[v]$  and  $u$  is a descendant of  $v$  in a DFS tree

Using the shorthand “ $(x$ ” for  $d[x]$  and “ $x)$ ” for  $f[x]$ ,  $(u\ u)\ (v\ v)$  and  $(u\ (v\ v)\ u)$  are possible (and the symmetric cases), but  $(u\ (v\ u)\ v)$  is not possible.

**Corollary 5.6:** Vertex  $v$  is a descendant of vertex  $u$  iff  $d[u] < d[v] < f[v] < f[u]$ .

*Proof:* Immediate from the parenthesis theorem. □

**Theorem 5.7** (White edge theorem): A vertex  $v$  is a descendant of  $u$  iff, at time  $d[u]$ , there exists a path from  $u$  to  $v$  consisting of entirely white vertices.

*Proof:* **TODO**

□

**Definition 5.8:** We can classify edges of the searched graph:

- **Tree edges**  $(u, v)$  are edges of the DFS forest;  $v$  is white when  $(u, v)$  is explored;
- **Back edges**  $(u, v)$  lead from a node to an ancestor in the DFS tree.  $v$  is grey when  $(u, v)$  is explored;
- **Forward edges** lead from a node  $u$  to a non-child descendant in the DFS tree, i.e. they lead to a descendant but are not edges in the DFS forest.  $v$  is black when  $(u, v)$  is explored;
- **Cross edges** do not lead to an ancestor or descendant; this could be between nodes in the same tree or in a different tree.  $v$  is black when  $(u, v)$  is explored.

**Remark:** We can link the parentheses of  $u, v$  to the classification of the edge  $(u, v)$ :

- If  $d[u] < d[v] < f[v] < f[u]$ , then  $(u, v)$  is either a tree edge or a forward edge.
- If  $d[v] < d[u] < f[u] < f[v]$ , then  $(u, v)$  is a back edge.
- If  $d[v] < f[v] < d[u] < f[u]$ , then  $(u, v)$  is a cross edge.
- $d[u] < f[u] < d[v] < f[v]$  cannot happen because if  $v$  is not discovered when  $u$  is discovered, and  $(u, v) \in E$ , then  $v$  would be explored before  $u$  is finished.

**Theorem 5.9:** A directed graph  $G$  has a cycle iff  $G$  has a back edge.

*Proof:*

$\Leftarrow$ : If  $(u, v)$  is a back edge, then there is a cycle consisting of the back edge and the path in the DFS tree from  $v$  to  $u$ .

$\Rightarrow$ : Suppose  $\langle v_0, \dots, v_k \rangle$  is a cycle, and suppose w.l.o.g. that  $v_0$  is discovered first in DFS. Then  $(v_k, v_0)$  is by definition a back edge. □

**Remark:** DAGs can be used to represent dependency problems; if  $A$  depends on  $B$ , draw an edge from  $B$  to  $A$ .

**Definition 5.10:** A **topological sort** of a DAG  $G := (V, E)$  is a total ordering of vertices,  $< \subseteq V \times V$ , such that if  $(u, v) \in E$  then  $u < v$ , and otherwise  $u \not< v$ .

TopologicalSort( $V, E$ ):

```
f[v | v in V] = DFS(V, E)
return V sorted according to decreasing f[v]
```

This has running time  $O(|V| + |E|)$ .

**Remark:** A topological sort gives us an order to complete tasks in the case where a DAG encodes dependencies between tasks.

**Proposition 5.11** (correctness of topological sort): Given a DAG  $G := (V, E)$ , if  $(u, v) \in E$  then  $f[u] > f[v]$ .

*Proof:* When  $(u, v)$  is explored,  $u$  is grey (because it has been discovered but not finished). Then by case distinction on the colour of  $v$ :

- If  $v$  is white, then  $v$  is a descendant of  $u$ , and by the parenthesis theorem,  $d[u] < d[v] < f[v] < f[u]$ .
- If  $v$  is black, then  $v$  is finished, but  $u$  is not yet, so  $f[v] < f[u]$ .
- $v$  cannot be grey, otherwise  $u$  would be a descendant of  $v$ , so  $(u, v)$  would be a backedge. By Theorem 5.9, there is a cycle in  $G$ , which is a contradiction because  $G$  is a DAG.

□

**Remark:** When DFS is applied to an undirected graph, the DFS trees correspond to the connected components of the graph. These can be identified by the discovery and finishing times.

**Definition 5.12:** In a digraph, two vertices  $u, v$  are **strongly connected** if there is a path from  $u$  to  $v$  and from  $v$  to  $u$ .

**Definition 5.13:** A **strongly connected component (SCC)** of a digraph  $G := (V, E)$  is a maximal set of vertices  $C \subseteq V$  such that for all  $u, v \in C$ ,  $u$  and  $v$  are strongly connected.

**Lemma 5.14:** Given distinct strongly connected components  $C, C'$  of a digraph  $G$ ,  $u, v \in C, u', v' \in C'$ , if there is a path from  $u$  to  $u'$  in  $G$ , then there is not a path from  $v'$  to  $v$  in  $G$ .

*Proof:* Suppose there is a path from  $u$  to  $u'$ . Then for every  $x \in C, y \in C'$ , there exists a path from  $x$  to  $y$  via the path connecting  $u$  and  $u'$ . If there were a path from  $v'$  to  $v$ , there would be a path from  $y$  to  $x$  via that path, so  $x$  and  $y$  would be strongly connected and  $C, C'$  would be part of the same SCC; contradiction. Therefore there is no path from  $v'$  to  $v$ . □

**Definition 5.15:** The **SCC graph** of a digraph  $G := (V, E)$  is the graph  $G^{\text{SCC}} := (V^{\text{SCC}}, E^{\text{SCC}})$ , such that

- $V^{\text{SCC}}$  has one vertex  $v_C$  for every SCC  $C$  in  $G$
- $(v_C, v_{C'}) \in E^{\text{SCC}}$  iff there exist  $u \in C, u' \in C'$  such that there is a path from  $u$  to  $u'$ .

**Remark:** From Lemma 5.14, it follows that any SCC graph is a DAG.

**Definition 5.16:** We extend discovery and finishing times for sets of vertices  $U \subseteq V$ :

- $d[U] := \min\{d[u] \mid u \in U\}$ , the earliest discovery time amongst  $U$ ;
- $f[U] := \max\{f[u] \mid u \in U\}$ , the latest finishing time amongst  $U$ .

**Definition 5.17:** We say that there is an edge between SCCs  $C, C'$  if  $\exists u \in C, u' \in C'$  s.t.  $(u, u') \in E$ .

**Lemma 5.18:** Let  $C, C'$  be distinct SCCs in  $G := (V, E)$ . If there is an edge from  $C$  to  $C'$ , then  $f[C] > f[C']$ .

*Proof:* This is equivalent to Proposition 5.11, because if there is an edge  $(u \in C, v \in C')$  from  $C$  to  $C'$ , then  $f[u] > f[v]$ ; we can choose  $u$  and  $v$  to have maximal finish times within their respective SCCs, so by our extended definition of finish time for sets of vertices,  $f[C] > f[C']$ .  $\square$

**Lemma 5.19:** If  $C \neq C'$  and  $f[C] < f[C']$ , then there cannot be an edge from  $C$  to  $C'$ .

*Proof:* Equivalent to Lemma 5.18 by contraposition.  $\square$

**Definition 5.20:** For a digraph  $G := (V, E)$ , the **transpose** of  $G$  is  $G^\top = (V, E^\top)$  where

$$E^\top = \{(u, v) \mid (v, u) \in E\}.$$

**Remark:** We can create  $G^\top$  in  $\Theta(|V| + |E|)$  time using adjacency lists.

**Theorem 5.21:**  $G$  and  $G^\top$  have the same SCCs.

**Remark:** For distinct SCCs  $C, C'$  in  $G$ , if  $f[C] > f[C']$ , then

- in  $G$  there cannot be an edge from  $C'$  to  $C$
- in  $G^\top$  there cannot be an edge from  $C$  to  $C'$ .

This means that running DFS on  $G^\top$  starting from the SCC with the largest finishing time, we will not find edges to any other SCC.

**Definition 5.22:** The **SCC algorithm**, or **Kosoraju's algorithm**, identifies all SCCs in a digraph  $G$ :

- run DFS on  $G$
- run DFS on  $G^\top$ , exploring in order of decreasing finishing time of the first DFS
- the trees in the DFS forest of the second DFS correspond to the SCCs of  $G$

## 6. Shortest paths in graphs

**Definition 6.1:** Consider a digraph  $G := (V, E)$  with **weight function**  $w : E \rightarrow \mathbb{R}$ .

The **weight** or length of a path  $p := \langle v_0, \dots, v_k \rangle$  is

$$w(p) := \sum_{i=0}^{k-1} w(v_i, v_{i+1}).$$

**Definition 6.2:** The **shortest-path weight** between two vertices  $u, v$  is

$$\delta(u, v) := \begin{cases} \min\{w(p) \mid p \text{ is a path from } u \text{ to } v\} & \text{if } \exists \text{ a path between } u \text{ and } v \\ \infty & \text{otherwise.} \end{cases}$$

**Definition 6.3:** A **shortest path** between two vertices  $u, v$  is a path  $p$  such that  $w(p) = \delta(u, v)$ .

**Definition 6.4:** A **FIFO queue** (first in, first out) is a data structure with the following operations:

- $\text{enqueue}(Q, x)$  - inserts  $x$  at the end of a queue
- $\text{dequeue}(Q)$  - removes and returns the item at the head of the queue
- $\text{isempty}(Q)$  - equivalent to  $Q \neq \emptyset$

All of these operations can be implemented in  $O(1)$  if using a linked list with a pointer to the start and end.

**Definition 6.5:** Considering first the simple case where all weights are equal to 1. Let  $\delta(u, v)$  be the minimum number of edges on a path from  $u$  to  $v$ , if such a path exists, otherwise  $\delta(u, v) = \infty$ .

Given a source vertex  $s$ , **breadth-first search (BFS)** finds all vertices  $v_i$ s reachable from  $s$ , the shortest path lengths  $\delta(s, v_i)$ , and the shortest paths from  $s$  to the  $v_i$ s.

Idea:

- send out waves of increasing length from the source  $s$
- when a vertex  $v$  is reached, put it in the (FIFO) queue  $Q$
- when all of  $v$ 's neighbours have been reached, remove  $v$  from  $Q$

The output is a distance  $d[v]$  and predecessor  $\pi[v]$  for every  $v \in V$ .

```

BFS(V, E, s):
  for v ∈ V \ { s }:
    d[v] = ∞
    π[v] = null
  d[s] = 0
  Q = ∅
  enqueue(Q, s)
  while Q != ∅:
    u = dequeue(Q)
    for v ∈ adj(u):
      if d[v] = ∞:
        d[v] = d[u] + 1
        π[v] = u
        enqueue(Q, v)

```

**Remark:** BFS takes time  $O(|V| + |E|)$ , because every vertex will be put in the queue (and extracted) exactly once, and the inner  $\text{adj}(u)$  loop will run exactly once for every edge across the whole running on the algorithm.

**Definition 6.6:** The **BFS tree** consists of vertices reachable from  $s$  and edges  $(u, v)$  where  $u = \pi[v]$ .

More formally, the BFS tree is the graph

$$G_\pi := (\{v \in V \mid \pi[v] \neq \text{null}\} \cup \{s\}, \{(u, v) \in E \mid u = \pi[v]\})$$

**Lemma 6.7:** **TODO** there exists a path of length  $d[v]$

**Corollary 6.8:** **TODO** lower bound on  $d$

**Lemma 6.9:** If  $u$  is enqueued before  $v$ , then  $d[u] \leq d[v]$ . Furthermore, if  $Q = \langle v_1, \dots, v_r \rangle$  is the queue at any given step of BFS, then

$$d[v_1] \leq d[v_2] \leq \dots \leq d[v_r] \leq d[v_1] + 1$$

*Proof:* **TODO** by induction □

**Lemma 6.10:** **TODO** upper bound on  $d$

**Theorem 6.11:**  $d[v] = \delta(s, v)$ .

*Proof:* **TODO** just follows from previous lemmas; also consider  $d[v] = \infty$ . □

**Theorem 6.12:** If  $0 < d[v] < \infty$ , then  $\pi[v]$  is the predecessor of  $v$  **TODO**

**Remark:** BFS uses a queue but DFS uses a stack (possibly implicitly).

**Definition 6.13:** **Dijkstra's algorithm** is essentially BFS for weighted graphs. It solves the “single-source shortest path” problem for non-negative weights.

It uses a min-priority queue  $Q$  rather than a FIFO queue, with keys given by the shortest-path weight estimates  $d[v]$ .

At termination:

- $d[v]$  is the distance from  $s$  to  $v$
- $\pi[v]$  is the predecessor of  $v$  on a shortest path from  $s$  to  $v$ , if such a path exists

Dijkstra( $V, E, w, s$ ):

```

for each  $v \in V$ :
     $d[v] = \infty$ 
     $\pi[v] = \text{null}$ 
 $d[s] = 0$ 
 $Q = \text{MakeMinQueue}(V)$  with  $d[v]$  as keys
while  $Q \neq \emptyset$ :
     $u = \text{ExtractMin}(Q)$ 
    for each  $v \in \text{adj}(u)$ :
        if  $d[u] + w(u, v) < d[v]$ :
             $d[v] = d[u] + w(u, v)$ 
             $\pi[v] = u$ 
             $\text{DecreaseKey}(Q, v, d[v])$ 
    
```

Invariants for the while loop:

$$I_1 := \forall v \in V . d[v] \geq \delta(s, v);$$

$$I_2 := \forall v \in S := V \setminus Q . d[v] = \delta(s, v).$$

**TODO** initialisation

Then suppose that  $I_1$  holds before an iteration of the while loop. For any vertex  $v \in V$ , either  $d[v]$  did not change, or it did change. If  $d[v]$  did not change, then  $d[v] \geq \delta(s, v)$  by  $I_1$ . Otherwise, let  $u$  be the vertex selected by `ExtractMin` in this iteration of the loop. Then we have that

$$\begin{aligned} d[v] &= d[u] + w(u, v) \\ &\geq \delta(s, u) + w(u, v) \text{ by } I_1 \\ &\geq \delta(s, v) \text{ because } \delta(s, u) + \mathbf{TODO} \end{aligned}$$

Now suppose that  $I_2$  holds before an iteration. Then for any  $v \in S'$ , either:

- $v \in S'$ , then  $d[v] = \delta(s, v)$  by  $I_2$
- $v \notin S$  and there is no path from  $s$  to  $v$ , then  $d[v] \geq \delta(s, v) = \infty \implies d[v] = \infty$
- $v \notin S$  and there is a path from  $s$  to  $v$ . Then there is also a shortest path  $s \xrightarrow{p} v$  **TODO**

**TODO** finish maintenance of  $I_2$

At termination,  $S = V$ , so  $\forall v \in V . d[v] = \delta(s, v)$ .

**TODO** remark about optimal substructure

**Lemma 6.14** (Convergence property): **TODO**

**Theorem 6.15:** For  $v \neq s, d[v] < \infty$ ,  $\pi[v]$  is the predecessor of  $v$  on a shortest path from  $s$  to  $v$ .

*Proof:* **TODO**

□

**Remark:** Total running time of Dijkstra's algorithm is  $O((|V| + |E|) \log |V|)$ , because `ExtractMin` ( $O(\log |V|)$ ) is executed  $|V|$  times, and `DecreaseKey` ( $O(\log |V|)$ ) is executed  $|E|$  times.

**Definition 6.16:** The **Bellman-Ford algorithm** takes a graph with weight function  $w : E \rightarrow \mathbb{R}$  (possibly negative), and returns `false` if there exists a negative-weight cycle reachable from  $s$ , otherwise `true`, along with  $d[v], \pi[v]$  for each  $v \in V$ .

```

BellmanFord(V, E, w, s):
  for each v ∈ V:
    d[v] = ∞
    π[v] = null
  d[s] = 0
  for i = 1 to |V| - 1: // run enough times to correctly compute distance
    for each (u, v) ∈ E:
      if d[u] + w(u, v) < d[v]: // Loop body the same as Dijkstra!
        d[v] = d[u] + w(u, v)
        π[v] = u
  for (u, v) ∈ E:
    if d[u] + w(u, v) < d[v]: // check for negative-weight cycles
      return false
  return true

```

This is  $O(|V||E|)$ .

**TODO** invariants

**TODO** full topological sort algorithm for DAGs to find shortest path

**Definition 6.17:** The **Floyd-Warshall algorithm** solves the **all-pairs shortest paths** problem for graphs with non-negative weights; that is, it gives the shortest path between every pair of vertices.

Suppose we have vertex set  $V = \{1, \dots, n\}$ .

Let  $d[i, j; k]$  be the length of the shortest path from  $i$  to  $j$ , all of whose intermediate nodes are in the interval  $[1, k]$ .

We initialise

$$d[i, j; 0] = \begin{cases} w(i, j) & \text{if } (i, j) \in E \\ \infty & \text{otherwise.} \end{cases}$$

Since there are no negative cycles, a shortest path from  $i$  to  $j$  using vertices in  $[1, k]$  goes through  $k$  at most once, so we use the recurrence

$$d[i, j; k + 1] = \min\{d[i, j; k], d[i, k + 1; k] + d[k + 1, j; k]\}.$$

This covers all possible routes because we consider all  $k$ , and choose to either pass through  $k$ , or not pass through  $k$ .

The algorithm is just dynamic programming as usual (**TODO** write it out anyway).

The running time is  $O(|V|^3)$ .

**Remark:** Some other options for all-pairs shortest paths:

- Run Bellman-Ford for every vertex as a source:  $O(|V|^2|E|)$ , which is  $O(|V|^3)$  for sparse graphs and  $O(|V|^4)$  for dense graphs.

- Run Dijkstra for every vertex as a source (if all weights not-negative):  $O(|V|\log|V|(|V| + |E|))$ , which is  $O(|V|^2 \log|V|)$  for sparse graphs and  $O(|V|^3 \log|V|)$  for dense graphs.

## 7. Greedy algorithms

**Definition 7.1:** In a **greedy algorithm**, at each step we greedily make the choice that offers the greatest immediate benefit (the **greedy choice**). This choice is not reconsidered at subsequent steps.

**Remark:** Dijkstra's algorithm is an example of a greedy algorithm.

**Remark:** The greedy approach doesn't always work, but when it does it's nice because it's simple and doesn't require keeping track of subproblem solutions.

**Definition 7.2:** Given a connected undirected graph  $G = (V, E)$  with weights  $w : E \rightarrow \mathbb{R}_{\geq 0}$ , the **minimum spanning tree (MST)** is the connected acyclic subgraph that has minimum weight and connects all the vertices of  $G$ . For positive weights, a minimum spanning tree always exists (because we can just look at all possible spanning trees and take the minimum).

We will identify a tree within  $G$  by its edge set  $T \subseteq E$ .

**Lemma 7.3:**

An undirected graph is a tree iff every pair of vertices is connected by a unique (simple) path.

*Proof:* A connected undirected graph has a cycle iff there exist two vertices connected by distinct paths. □

**Lemma 7.4:** If a graph  $G := (V, E)$  is a tree, then  $|E| = |V| - 1$ .

*Proof:*

$G$  connected  $\implies |E| \geq |V| - 1$

$G$  acyclic  $\implies |E| \leq |V| - 1$ .

TODO more formally

□

**Definition 7.5:** A **spanning tree** of a graph  $G := (V, E)$  is a subgraph with edge set  $T \subseteq E$  such that  $T$  is a tree, and  $T$  reaches all vertices of  $G$ .

**Remark:** A spanning tree is:

- a minimal connected subgraph (removing an edge disconnects it)
- a maximal acyclic subgraph (adding an edge creates a cycle).

Therefore any spanning tree has exactly  $|V| - 1$  edges.

**Lemma 7.6:** Every connected graph has a spanning tree.

*Proof:* TODO

□

**Definition 7.7:** A **minimal spanning tree (MST)** is a spanning tree of minimal weight, where the weight  $w(T)$  of a tree  $T$  is the sum of the weights of all the edges of  $T$ .

**Remark:** A graph can have multiple MSTs, but they must all have the same number of edges (because they are spanning trees).

The general algorithm for building an MST:

- start from  $A = \emptyset$ ; this is a (trivial) subset of an MST
- Add edges to  $A$ , maintaining that  $A$  is a subset of an MST
- Stop when no edge can be added to  $A$  any more, then  $A$  is an MST.

**Definition 7.8:** Let  $A \subseteq E$  be a subset of an MST  $T$ . We say that an edge  $(u, v)$  is **safe** to add to  $A$  iff  $A \cup \{(u, v)\}$  is a subset of some MST.

**Definition 7.9:** A **cut** is a partition of the vertex set into  $S$  and  $V \setminus S$ .

An edge  $(u, v) \in E$  **crosses** a cut if one endpoint is in  $S$  and the other is in  $V \setminus S$ .

A cut **respects**  $A \subseteq E$  if no edge in  $A$  crosses the cut.

An edge crossing a cut is **light** if its weight is minimal over all edges that cross that cut.

**Lemma 7.10** (Cut Lemma): Let  $A$  be a subset of some MST. If  $(S, V \setminus S)$  is a cut that respects  $A$ , and  $(u, v)$  is a light edge crossing the cut, then  $(u, v)$  is safe for  $A$ .

*Proof:* Let  $T$  be an MST that includes  $A$ . Since  $T$  is a tree, it contains a unique path  $P$  between  $u$  and  $v$ .

Path  $P$  must cross the cut  $(S, V \setminus S)$  at least once, because  $u$  and  $v$  are on different sides of the cut, so at some point an edge must cross the cut. Let  $(x, y)$  be an edge of  $P$  that crosses the cut. Adding  $(u, v)$  and deleting  $(x, y)$  creates a tree  $T'$ .

$T'$  is a tree because we create a cycle and then remove one edge from it, keeping the graph connected.  $T'$  is minimal because the weight of  $T'$  is  $w(T') = w(T) - w(x, y) + w(u, v) \leq w(T)$ , because we chose  $(u, v)$  to be light (i.e. having minimal weight of all the edges that cross the cut).  $T'$  contains  $A$  because  $A$  was included in  $T$ , and  $A$  did not contain  $(x, y)$ , because the cut respected  $A$ .  $\square$

**Definition 7.11:** A **disjoint-set data structure** keeps track of a family  $\mathcal{S}$  of dynamic disjoint sets  $S_1, \dots, S_k$ . Each set is identified by some arbitrary representative.

It has three operations:

- **MakeSet**( $x$ ) adds  $\{x\}$  to  $\mathcal{S}$
- **FindSet**( $u$ ) returns the representative of the set containing  $u$
- **Union**( $x, y$ ) removes  $S_x, S_y$  from  $\mathcal{S}$  and adds  $S_x \cup S_y$  to  $\mathcal{S}$ .

A good representation is as a disjoint-set forest, when elements are stored in an array with pointers to a parent leading to a root of each set. Keeping this balanced gives amortised complexity  $O(m\alpha(n))$  where  $m$  is the number of operations and  $n$  is the number of **MakeSet** operations, and  $\alpha(n)$  is the inverse Ackermann function, which is extremely slowly-growing and is at most 4 for any feasible inputs.

**Definition 7.12:** **Kruskal's algorithm:**

Start from  $A = \emptyset$ . At each step, pick the edge with the smallest weight and add it to  $A$  if it does not create a cycle.

To avoid cycles, we need to keep track of the connected components, for instance by using a disjoint-set data structure.

```
Kruskal(V, E, w):
  A =  $\emptyset$ 
  for v  $\in$  V:
    MakeSet(v)
  sort E into increasing order by weight w
  for each edge (u, v) from sorted edge list:
    if FindSet(u)  $\neq$  FindSet(v):
      A = A  $\cup$  {(u,v)}
      Union(u, v)
  return A
```

There are  $|V|$  MakeSet operations, and  $\Theta(|V| + |E|) = \Theta(|E|)$  total disjoint-set operations. So the disjoint-set operations take  $O(|E|\alpha(|V|))$  (for a good disjoint-set implementation). So the sorting of the edges dominates. Therefore Kruskal's algorithm has time complexity  $O(|E| \log |E|)$ .

**Definition 7.13:** **Prim's algorithm** picks a vertex  $r \in V$  and grows the tree from that vertex.

We set  $S = \{r\}$  and  $A = \emptyset$ . Then at every step, find a light edge  $(u, v)$  connecting  $u \in S$  to  $v \in V \setminus S$ . Update  $S$  to  $S \cup \{v\}$  and  $A$  to  $A \cup \{(u, v)\}$ .

We use a min-heap/priority queue  $Q$  such that  $Q = V \setminus S$ , the key of  $v$  is the minimum weight of any edge  $(u, v)$  where  $u \in S$ , and if no such edge exists set the key to be  $\infty$ .

To find a light edge crossing the cut  $(S, V \setminus S)$ , take the minimum from the queue. If  $v = \text{ExtractMin}(Q)$ , then there exists a light edge  $(u, v)$  for some  $u \in S$ . The vertex  $u$  can be retrieved by a backpointer: when the key of  $v$  is to  $w(u, v)$ , we define  $\pi[v] = u$ .

```
Prim(V, E, w, r):
  for u in V:
    key[u] = ∞
    π[u] = null
    Insert(Q, u)
  DecreaseKey(Q, r, 0)
  while Q != ∅:
    u = ExtractMin(Q)
    for each v in Adj[u]:
      if v in Q and w(u,v) < key[v]:
        π[v] = u
        DecreaseKey(Q, v, w(u,v))
```

Then  $\pi$  gives us the edges.

This has running time  $O(|E| \log |V|)$  because **DecreaseKey** ( $O(\log |V|)$ ) is executed exactly once for every edge.

**Remark:** Because  $\log |E| = O(\log |V|)$  for a connected graph, Kruskal and Prim have the same asymptotic running time.

**Remark:** Prim can be improved to  $O(|E| + |V| \log |V|)$  using a better min-priority queue implementation.

**Definition 7.14:** The **activity selection problem** is, given a set of activities  $(s_i, f_i)$ ,  $i = 1, \dots, n$ , to select a maximum-size subset of activities that do not overlap. (Where  $s_i$  is the start time and  $f_i$  is the finish time).

We can reduce this to a graph where edges between nodes indicate a scheduling conflict, and try to find a maximal independent set of vertices.

We could try to find this by greedily selecting vertices of minimal degree. However, this wouldn't always work:

**TODO** ( one node at bottom, connected to 4 in next row, connected to four in row above which form a clique )

It turns out that this is an NP-hard problem (i.e. we can verify a solution in polynomial time, but finding a solution is difficult).

However, this doesn't mean that activity selection is hard, because not all graphs are an activity selection problem. For example, a graph with an induced cycle  $> 3$  cannot be an activity selection problem. **TODO** image

**Lemma 7.15:** There exists an optimal solution to an activity selection problem that contains an activity with minimum finish time.

*Proof:* Let  $a$  be an activity with minimum time. Consider an optimal solution and assume it does not contain any activity with the same finish time as  $a$ . By removing the activity of the optimal solution with minimum finish time and adding  $a$ , we create a valid solution that has the same number of activities.  $\square$

**Corollary 7.16:** The greedy algorithm whereby we take the activity with minimal finish time at each step always gives an optimal solution to the activity selection problem.

```

ActivitySelection(s, f):
  sort the activities in order of increasing f
  A = { 1 }
  k = 1
  for j = 2 to n:
    if s[j] >= f[k]:
      A = A  $\cup$  { j }
      k = j
  return A

```

**Remark:** The runtime is dominated by the sorting, so is  $O(n \log n)$ .

## 8. Stable matching

**Definition 8.1:** A **matching** is an undirected graph where all connected components are pairs of vertices.

Equivalently, a matching  $M$  is a set of ordered pairs  $(h, s)$  with  $h \in H$  and  $s \in S$ , such that **TODO**

Goal: given a set of preferences among hospitals and med students, design a self-reinforcing admissions process.

**Definition 8.2:** Hospital  $h$  and student  $s$  form an **unstable pair** if:

- $h$  prefers  $s$  to one of its admitted students
- and  $s$  prefers  $h$  to its assigned hospital.

**Definition 8.3:** A **stable assignment** is an assignment with no unstable pairs.

This is the desirable condition. Note that individual self-interest prevents any hospital-student side deal (if there are any unstable pairs though, the hospital and student could both complain).

**Definition 8.4:** A **perfect matching** is **TODO**

**Definition 8.5:** A **stable matching** is a perfect matching with no unstable pairs.

**TODO** Gale-Shapely deferred acceptance algorithm

**Remark:** This is  $O(n^2)$ .

**Lemma 8.6:** The Gale-Shapley algorithm finds a perfect matching.

*Proof:* Suppose for the sake of contradiction that some hospital  $h$  is unmatched at termination. Then some student, say  $s$ , is unmatched at termination. This means that  $s$  was never proposed to, but  $h$  must have proposed to every student, because it is unmatched. Contradiction, so the matching upon termination must be perfect.  $\square$

**Lemma 8.7:** The matching  $M$  produced by the Gale-Shapley algorithm is stable.

*Proof:* Consider a pair  $(h, s)$  that is not in  $M$ .

Either:

- $h$  proposed to  $s$ , in which case  $h$  prefers its student in  $M$  to  $s$
- $h$  proposed to  $s$ : therefore  $s$  rejected  $h$  at some point, so  $s$  ended up with a more preferred hospital.

**TODO:** why does this mean it's stable?

□

**Definition 8.8:** A student  $s$  is a **valid partner** for hospital  $h$  if there exists any stable matching in which  $h$  and  $s$  are matched.

**Remark:** The Gale-Shapley algorithm gives the hospital-optimal assignment, i.e. each hospital received the best valid partner. As a corollary, we get that the hospital-optimal assignment is stable.

This is also the student-pessimal assignment (each student gets the worst valid partner). As a corollary, we get that the student-pessimal assignment is stable.

**Remark:** A student can get a better outcome by lying about their preferences.

## Index

Activity selection problem .....	29	Max-heap .....	10
Adjacency matrix .....	15	Max-heap property .....	11
Algorithm .....	2	MaxHeapify .....	11
All-pairs shortest paths .....	25	Minimal spanning tree (MST) .....	27
Asymptotic lower bound .....	4	Minimum spanning tree (MST) .....	26
Asymptotic tight bound .....	4	Optimal substructure .....	13
Asymptotic upper bound .....	2	Perfect matching .....	31
BFS tree .....	22	Prim's algorithm .....	29
Back edges .....	18	Priority queue .....	13
Bellman-Ford algorithm .....	24	Respects .....	27
Binary search .....	8	SCC algorithm .....	21
Breadth-first search (BFS) .....	22	SCC graph .....	20
Counting sort .....	9	Safe .....	27
Cross edges .....	18	Select .....	8
Crosses .....	27	Shortest path .....	21
Cut .....	27	Shortest-path weight .....	21
DFS forest .....	17	Spanning tree .....	27
DFS trees .....	17	Stable assignment .....	31
Depth-first search (DFS) .....	16	Stable matching .....	31
Descendant .....	17	Strassen's algorithm .....	7
Dijkstra's algorithm .....	23	Strongly connected .....	19
Disjoint-set data structure .....	28	Strongly connected component (SCC) .	19
Divide and conquer algorithm .....	4	Topological sort .....	18
Dynamic programming .....	13	Transpose .....	20
Efficient .....	2	Tree edges .....	18
FIFO queue .....	21	Unstable pair .....	31
Fast Fourier Transform (FFT) .....	9	Valid partner .....	32
Floyd-Warshall algorithm .....	25	Weight .....	21
Forward edges .....	18	Weight function .....	21
Greedy algorithm .....	26		
Greedy choice .....	26		
Heap .....	10		
Heap sort .....	12		
Height .....	9, 11		
IncreaseKey .....	13		
Insertion sort .....	2		
Ith-order statistic .....	8		
Karatsuba algorithm .....	6		
Karatsuba-Ofman algorithm .....	6		
Kosoraju's algorithm .....	21		
Kruskal's algorithm .....	28		
Light .....	27		
MakeMaxHeap .....	12		
Matching .....	30		