2 Sorting

Given two functions f and g, both $\mathbb{N} \to \mathbb{R}$, we say $f(n)$ is $O(g(n))$ if

$$\exists \kappa > 0 \text{ and } n_0 \in \mathbb{N} \text{ such that } \forall n \geq n_0, |f(n)| \leq \kappa |g(n)|$$

and we say $f(n)$ is $\Omega(g(n))$ if

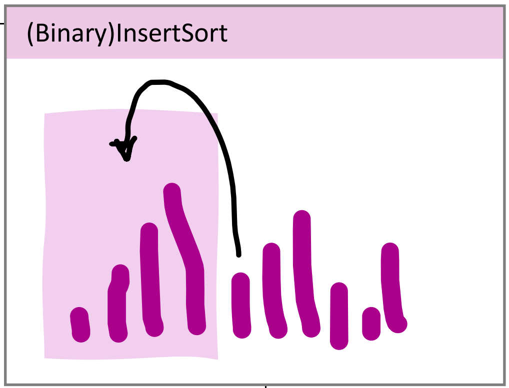
$$\exists \delta > 0 \text{ and } n_0 \in \mathbb{N} \text{ such that } \forall n \geq n_0, |f(n)| \geq \delta |g(n)|.$$

If $f(n)$ is $O(g(n))$ and also $\Omega(g(n))$ we say that $f(n)$ is $\Theta(g(n))$.

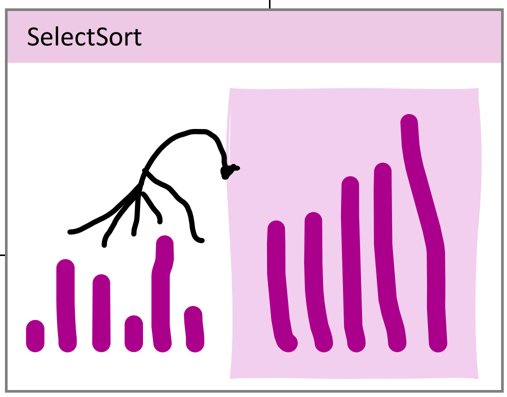
```
def insertsort(x):
  for i in 1..(len(x)-1):
    # assert x[0:i] is sorted
    j = i - 1
    while j >= 0 and x[j] > x[j+1]:
      swap x[j] with x[j+1]
      j = j - 1
    # assert x[0:i+1] is sorted

# Same thing, more succinctly
def insertsort(x):
  for i in 1..(len(x)-1):
    do a linear search for where x[i] should go, and insert it
    there

def binaryinsertsort(x):
  for i in 1..(len(x)-1):
    do a binary search for where x[i] should go, and insert it
    there
```

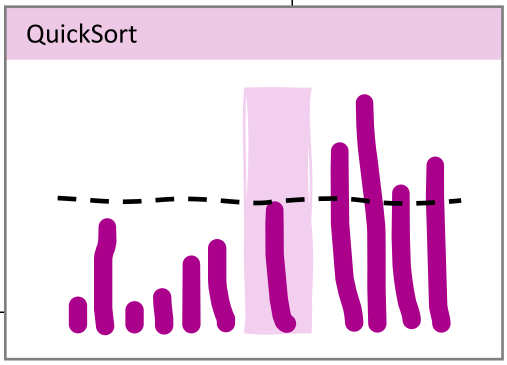



```
def selectsort(x):
  # This code fills in from the left,
  # the picture shows filling in from the right
  for i in 0..(len(x)-2):
    # Find what belongs at x[i]
    j = arg min_{i≤k<len(x)} x[k]
    swap x[i] with x[j]
```

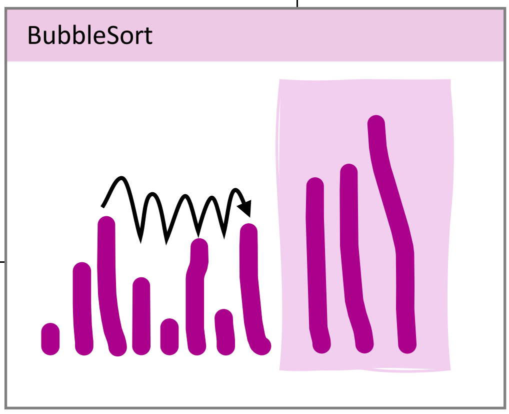


```
def quicksort(x):
  1.  Pick the last item to be the pivot, p = x[len(x) − 1].
  2.  Partition the array, so that
      it has the form
                    (items ≤ p) :: p :: (items ≥ p)
  3.  The pivot p is now in its correct place. Call quicksort on
      the left portion, and on the right portion.

def partition(x, p):
  i = just before first item
  j = just before p
  while True:
    while i < j and x[i] <= p: i++
    while i < j and x[j-1] >= p: j--
    if i < j:
      swap x[i] with x[j-1]
      i++, j--
  swap p with x[j]
```

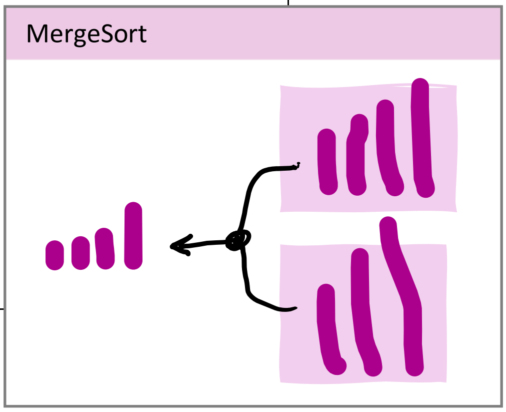



```
def bubblesort(x):
  while True:
    any_swaps = False
    for i in 0..(len(x)-2):
      if x[i] > x[i+1]:
        swap x[i] with x[i+1]
        any_swaps = True
    if not any_swaps:
      break
```



```
def mergesort(src, dst):
  n = len(src)
  If n==1, just copy src[0] into dst[0]. Otherwise:
  m = int(n/2)
  x1 = new array of length m
  mergesort(src=src[0:m], dst=x1)
  x2 = new array of length n-m
  mergesort(src=src[m:n], dst=x2)
  merge x1 and x2 into dst
  free x1 and x2

def merge(x1, x2, dst):
  # assert len(dst) == len(x1)+len(x2)
  i1,i2 = 0,0
  for j in 0..(len(dst)-1):
    dst[j] = min(x1[i1], x2[i2])
    advance i1 or i2 appropriately
```



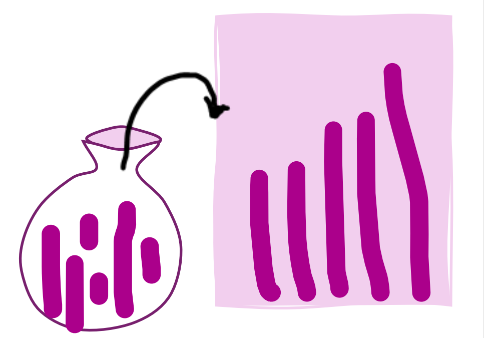
```
def heapsort(x):
  n = len(x)
  # Create the initial heap
  for i in 1..n-1:
    # assert x[0:i] is a heap
    add x[i] to heap and re-heapify
  # assert x[0:n] is a heap
  for i in n..1:
    # assert x[i:n] has largest n-i
    # assert x[0:i] is a heap
    swap x[0] with x[i-1]
    re-heapify x[0:i-1]

# Re-heapify by bubbling up from i
j = i
while j > 0 and x[j] > x[parent(j)]:
  swap x[j] with x[parent(j)]
  j = parent(j)

# Re-heapify by bubbling down from 0
j = 0
while x[j] < max(x[child1(j)], x[child2(j)]):
  swap x[j] with larger child
  j = larger child

# Faster way to create the initial heap
for i in ⌊n/2⌋..0:
  # assert trees rooted at (i+1)..n are heaps
  re-heapify the tree rooted at x[i]
  by bubbling down
```

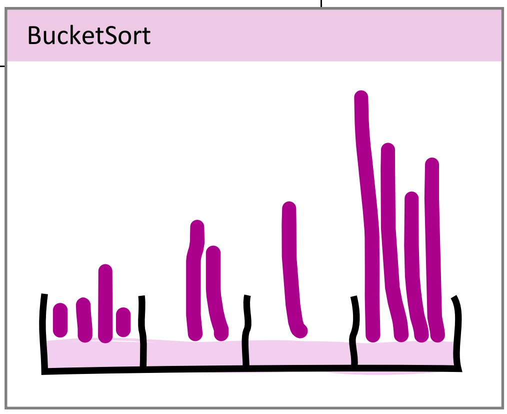
HeapSort



```
def radixsort(x):
  for each digit d, starting from
  the least significant:
    stably sort x by digit d
    # assert x is in order with
    # respect to digits d:end
```

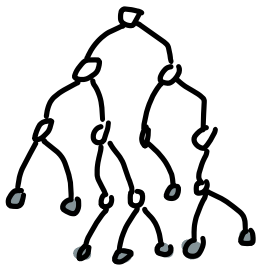
```
def countingsort(x, m):
  # Count num.occurrences of each value
  counts = …
  # Figure out the first location for each possible value
  nextpos = …
  y = new array of same size as x
  # Go through x and place each item into its
  # correct location
  for each value v in x:
    y[nextpos[v]] = v
    nextpos[v] += 1
  return y
```

```
def bucketsort(x, a):
  B = ⌈len(x)/a⌉
  buckets = array of B empty linked lists
  for each item v in x:
    append v to bucket ⌊key(v) × B⌋
  # assert: average number of items in each bucket is ≈a
  for each bucket:
    sort it with a O(n^2) algorithm
    output its values
```



3 Dynamic programming

We're given an initial state x_0, and we wish to choose a sequence of actions $[a_0, a_1, \dots]$. If we're in state x and we take action a, we gain reward $r_{x,a}$ and we move to next state $n_{x,a}$ (unless x is a terminal state, where no further actions are possible, in which case we gain reward t_x). What is the maximum possible total reward, starting from our initial state x_0?



Let $v(x)$ be the total reward that can be gained starting in state x. Then

$$v(x) = \begin{cases} t_x & \text{if } x \text{ is terminal} \\ \max_{a \in A} \{ r_{x,a} + v(n_{x,a}) \} & \text{otherwise} \end{cases}$$

Is it worth doing cardio?
Suppose we have a fixed number of total lifetime heartbeats. Each day we can choose to exercise or not. Let $x = (r, b)$ be our current state, where r is resting heart rate and b is the number of lifetime heartbeats remaining. If we exercise,

$$r \leftarrow r - \lambda(r - 50) \text{ and } b \leftarrow b - 23 \cdot 60 \cdot r - 60 \cdot 155$$

and if we don't exercise then

$$r \leftarrow r + \lambda(90 - r) \text{ and } b \leftarrow b - 24 \cdot 60 \cdot r$$

Should we exercise, and if so how often?

Rod cutting.

A DIY supplier has a steel rod of length $n \in \mathbb{N}$, and a machine that can cut it into smaller pieces. Different lengths can be sold for different prices; a piece of length $\ell \in \mathbb{N}$ fetches price p_ℓ. How should it be cut, to maximize profit? (The cut below, of a rod of length 10, fetches $£8 + £9 + £8 = £25$ and is sub-optimal.)

length	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
price	£1	£5	£8	£9	£10	£17	£17	£20	£24	£30

Matrix multiplication order.
The cost of multiplying two matrices depends on their dimensions: it takes ℓmn operations to perform the multiplication

$$\underset{\ell\times m}{A} \cdot \underset{m\times n}{B} = \underset{\ell\times n}{C}$$

If we want to compute the product of several matrices, we have a choice about the order of multiplication (because matrix multiplication is associative). For example, $ABCDE = (AB)((CD)E) = A(B((CD)E))$.

What is the least-cost way to compute $A_0A_1\cdots A_{n-1}$ where A_i has dimension $d_i \times d_{i+1}$?

Longest common subsequence.

A *subsequence* of a string s is any string obtained by dropping zero or more characters from s. Given two strings s and t, what's the longest subsequence they have in common? (The illustration shows a common subsequence of length 3, "HER", but it's not the longest common subsequence.)

T	H	E	B	A	R	B	I	E	M	O	V	I	E

O	P	P	E	N	H	E	I	M	E	R

Resource allocation.
Several different university societies have all requested to book the sports hall, request k having start time $u_k \in \mathbb{R}$ and end time $v_k \in \mathbb{R}$. The hall can only fit one activity at a time. What is the maximum number of requests that can be satisfied without overlap?

Alternative formulation: Let $t_0 < t_1 < \cdots < t_n$ be a sequence of distinct timepoints, and let request k have start time t_{U_k} and end time t_{V_k} where $U_k, V_k \in \mathbb{N}$.