COMP 3711/3711H– Fall 2020
Solutions to Collected Tutorial Questions

Version of November 9, 2021

M. J. Golin – HKUST

This document contains solutions to a collection of problems, many of which will be covered in the COMP3711 or COMP3711H tutorials.

A separate document containing only the descriptions of the problems, without solution, is also available.

A few problems do not have solutions provided. Individual solutions to those problems, in presentation slide format, are available on the class tutorial web page.

Please check the class tutorial web page to see which problems are assigned to which tutorial session.

We strongly recommend that you try solving the problems yourself BEFORE reading the answers or listening to the TA's presentation.

Note: Be aware that COMP 3711 and COMP3711H teach a very small number of topics slightly differently, e. g., the analysis of randomized Quicksort and the mathematical formulation for Max-Flow. A few tutorial problems on these subjects are oriented towards one or the other of the two classes.

The problems are roughly divided into the following topics

(SS) Sorting and Searching

(R) Randomization

(DC) Divide & Conquer

(GY) Greedy

(GR) Graphs

(DP) Dynamic Programming

(MM) Max Flow & Matchings

(HA) Hashing

Sorting and Searching (SS)

(SS1) $a_1, a_2, \ldots, a_n$ is a sequence that has the following property:
There exists some k such that

$$\forall\ i\ :\ 1 \le i < k, \quad a_i > a_{i+1} \qquad \forall\ i\ :\ k \le i < n, \quad a_i < a_{i+1}.$$

Such a sequence is *unimodal* with unique minimum a_k.
Design an $O(\log n)$ algorithm for finding k.

Solution:

Example: $A = [10, 8, 6, \mathbf{5}, 25, 30, 40, 70, 90, 100]$.

We can binary search for the transition point *where the items stop decreasing and start increasing.*

- *Define new array $B = [1, n-1]$ where*

$$B[i] = \begin{cases} + & \textit{if } A[i] > A[i+1], \\ - & \textit{if } A[i] < A[i+1]. \end{cases}$$

  Example: For A above, corresponding B is $B = [+, +, +, -, -, -, -, -, -]$.

- *No need to actually build $B[\,]$. Testing whether $B[i] = +$ or $-$ can be done by just testing whether $A[i] > A[i+1]$ or not.*
- *Unimodality implies that B is in form $B = [+, +, +, \ldots, +, -, - \ldots, -]$, where* **k**, *the location of the minimum value in A, is the location of the first "$-$" in B.*
- **k** *can therefore be found using an $O(\log n)$ binary search for the first "$-$" in $B[]$.*

(SS2) You are given an (implicit) infinite array $A[1, 2, 3.....]$.
and are told that, for some unknown n, the first n items in the array are positive integers sorted in increasing order while, for $i > n$, $A[i] = \infty$.

Give an $O(\log n)$ algorithm for finding the largest non-∞ value in A.

Solution: *Example:* $A = [1, 3, 4, 18, 25, \infty, \infty, \infty, \infty, \ldots]$.

If we knew some $m > n$, such that $A[m] = \infty$, binary search *could find n in ∞ in $O(\log m)$ time. But, we don't know such an m in advance.*

- *The trick is to use a* doubling search.
  Evaluate $A[1]$, $A[2]$, $A[2^2]$, $A[2^3]$, $A[2^4]$, $\ldots$, in that order
  until finding the first k such that $A[2^k] = \infty$.
- *$2^k > n$, so set $m = 2^k$ and run the binary search.*
- *By definition, $2^{k-1} \le n$.*
  $\Rightarrow \quad m = 2^k \le 2n \quad \Rightarrow \quad k = \log_2 m \le 1 + \log_2 n.$
- *Doubling search uses $O(k) = O(\log n)$ time to find k,*
  and
  the binary search step uses $O(\log m) = O(\log n)$ time to find n
  $\Rightarrow$ the entire procedure uses $O(\log n)$ time.

(SS3) Consider the heap implementation of a Priority Queue shown in class that keeps its items in an Array $A[]$.

Let Decrease-Key(i, x) be the operation that compares x to $A[i]$:

If $x \geq A[i]$ it does nothing.
If $x < A[i]$, it sets $A[i] = x$ and, if necessary, fixes $A[]$ so that it remains a Heap.

Show how to implement Decrease-Key(i, x) in $O(\log n)$ time, where n is the number of items in the Heap.

Note: We will use the operation Decrease-Key(i, x) later in the semester.

(SS4) (a) Illustrate how Mergesort would work on input [1,2,3,4,5,6,7,8,9].

(b) Illustrate how Mergesort would work on input [9,8,7,6,5,4,3,2,1].

(c) Illustrate how Quicksort would work on input [1,2,3,4,5,6,7,8].
Assume that the last item in the subarray is always chosen as the pivot.

(d) Illustrate how Quicksort would work on input [8,7,6,5,4,3,2,1].
Assume that the last item in the subarray is always chosen as the pivot.

(SS5) Your input is k sorted lists that need to be merged into one sorted list. The "obvious" solution is to modify the merging procedure from mergesort; at every step, compare the smallest items from each list and move the minimum one among them to the sorted list.

Finding the minimum value requires $O(k)$ time so, if the lists contain n items in *total* the full k-way merge would take $O(nk)$ time.

This can be solved faster.

Design an $O(n \log k)$-time algorithm to merge k sorted lists into one sorted list by making use of priority queues.

Note that each sorted list may contain a different number of elements.

Solution:

The obvious method is to walk through the lists, maintaining a pointer to the first (smallest) unmerged element in each list. At the start, the pointers point to the first element of each list. We'll call these items the heads *of the lists.*

Create a set S (of size k) containing the current head of each list.
Note that finding the minimum in this set uses $O(k)$ time.

Let A be the merged list being built. Start with A being empty.

At each step (including the first) find the current smallest head in S in $O(k)$ time.
Remove it from S and insert it at the end of the current merged list A.
Move the old head that had pointed to that smallest item one space to the right.
Add the new item it points to to S.
Continue until all items have been found.
This uses $O(nk)$ time.

Now modify the previous algorithm by keeping the set S of list heads in a priority queue Q.

Each step now requires extracting the minimum item x from Q from the priority queue and inserting the new smallest item from the list that x belonged to into Q (note that if x was the largest item on that list the algorithm does nothing).

Since each priority queue operation requires $O(\log k)$ time and the algorithm performs $O(n)$ inserts and $O(n)$ extract mins, the entire merge requires $O(n \log k)$ time.

(SS6) The following is the pseudo-code for a procedure known as *BubbleSort* for sorting an array of n integers in ascending order.

```
repeat
    swapped := false;
          for i = 1 to n-1
               if A[i] > A[i+1]  then
                  swap A[i] and A[i+1]
                  swapped := true
until  not(swapped)
```

(a) Prove that Bubble sort correctly sorts its input.

(b) What is the worst-case input for bubble sort? Use it to derive a lower bound on the time complexity of Bubblesort in the worst case.

Solution:

Claim 1: *When Bubble Sort terminates, the array must be sorted.*

Proof: *The algorithm terminates only if the last pass did not swap any pair, i.e*

$$A[1] \le A[2] \le \cdots \le A[n-1] \le A[n],$$

which means that the array is sorted.

Next observe

Claim 2: *After the i'th pass, the items in locations $A[n-i+1,\ldots,n]$ are in their proper sorted locations and none of them ever move again.*

Proof:

- *This is obviously true if $n = 1$ or $n = 2$.*
- *After the 1st pass, the largest element must be in location $A[n]$, and that item never moves after the first pass completes.*
- *After the 1st pass completes, it is as if Bubblesort is being run on array $A[1\ldots n-1]$.*
- *The claim then follows by induction.*

Combining claims (1) and (2) prove that Bubblesort correctly terminates with sorted input.

Claim 2 also immediately implies

Claim 3: *Bubble Sort on n items terminates after at most $n-1$ passes.*

Claim 2 further implies

Claim 4: *The ith pass in Bubble Sort performs at most $n-i$ swaps.*

Proof: *After the $(i-1)$'st pass, items in locations $A[n-i+2,\ldots,n]$ never move again. Thus, the i'th pass only works on items $A[1,\ldots,n-i+1]$ and can only perform at most $n-i$ swaps.*

From claim 4, the total number of swaps that can be performed is then at most

$$\sum_{i=1}^{n}(n-i)=\sum_{j=1}^{n-1}j=\frac{n(n-1)}{2}.$$

In the other direction note that if $A[\,]$ is sorted in decreasing order, i.e.,

$$A[1]>A[2]>\cdots>A[n-1]>A[n],$$

then pass i does exactly *$n-i$ swaps and Bubblesort will perform exactly*

$$\sum_{i=1}^{n}(n-i)=\sum_{j=1}^{n-1}j=\frac{n(n-1)}{2}$$

swaps. A reverse sorted array is therefore the worst case input.

This means that $\Theta\left(\frac{n(n-1)}{2}\right)=\Theta(n^2)$ is a tight bound on the time complexity of Bubblesort in the worst case.

(SS7) An array $A[1\ldots n]$ is circularly sorted if there exists some $k\in[1\ldots n]$ such that $A[k\ldots n]$ concatenated to $A[1\ldots k-1]$ is sorted. As an example

$$A=[30,40,55,10,18,24,27,28]$$

is circularly sorted with $k=4$.

How can you modify the binary search algorithm to search for an item in A in $O(\log n)$ time.

Note: To simplify the problem, assume that all of the items in A are unique (no duplicates).

Solution:

The problem is to search for key K.

For simplicity, we assume there are no repeated items.

Let $s=k$ be the index of the start of the array. In the example, $s=4$, with $A[s]=10$.

(a) *First suppose that s is known.*

Then you could just binary search for K separately in $A[1\ldots s-1]$ and also in $A[s\ldots n]$. This solves the problem in a total of $O(2\log_2 n)=O(\log n)$ time.

Note This can be made more practically efficient by first checking if $K\ge A[1]$ or not. If yes,, then binary search for K in $A[1\ldots s-1]$; if no, then binary search for K in $A[s\ldots n]$. This is also requires $O(\log n)$ time.

(b) *So, if s could be found in $O(\log n)$ time then, from part (a), full problem could be solved in $O(\log n)$ time. Now note the following properties.*

(i) *$s = 1$ if and only if $A[1] < A[n]$.*

(ii) *If $s \neq 1$ then*
If $[(i < j)$ AND $(A[i] > A[j])] \Rightarrow i < s \leq j$.

From (b) we can first check in $O(1)$ time whether $s = 1$ and, if not, use the modification of binary search below to find s :

```
1. $i = 1$; $j = n$                          % $i < j$ and $A[i] > A[j]$
2. While $j > i + 1$ do                       % Maintains invariant that $i < j$ and $A[i] > A[j]$
3.         $m = \lfloor \frac{i+j}{2} \rfloor$                  % $i < m < j$
4.         If $A[m] > A[i]$ then
5.             $i = m$
6.         Else
7.             $j = m$
8. $s = j$                                    % $i < s \leq j = i + 1$
```

Set $i = 1$; $j = n$. For these values, $i < j$ and $A[i] > A[j]$.

Next note that, from b(ii), this code **maintains** *the correctness of* the invariant *that, after every step, $i < j$ and $A[i] > A[j]$. Algorithm terminates with $i < s \leq j = i+1$, so $s = j$.*

At every step it halves the value of $j - i$. Thus, after $O(\log_2 n)$ steps, $j = i + 1$ and $s = j$.

To solve the full problem, first spend $O(\log n)$ time finding s and then use part (a) to binary search for K in $O(\log n)$ time.

(SS8) **Heapify**

In class, we learned how to maintain a min-heap implicitly in an array.
Given that $A[i \ldots (j-1)]$ represents an implicit min-heap, we saw how to add $A[j]$ to the min-heap , in $O(\log j)$ time.
This led to an $O(n \log n)$ algorithm for constructing a min-heap from array $A[1, \ldots, n]$.

For this problem show how to construct a min-heap from an array $A[1 \ldots n]$ in $O(n)$ time.

It might help to visualize the min-heap as a binary tree and not an array.

For simplification, you may assume that $n = 2^{k+1} - 1$ for some k, i.e., the tree is complete.

Hint: Consider "heapifying" the nodes processing them from bottom to top.

Solution: *"Heapify" the nodes row by row, moving from $h = 1$ to $h = k$ (h being the height of the node, with leaves having $h = 0$ and root having $h = k$).*

By "heapify' a node" we mean make the tree rooted at the node have the min-heap property.

Note that at the time we process a node, its two subtrees have already been heapified. Thus, we can heapify by bubbling the node down as many levels as appropriate (as shown in class when performing Extract-Min).

The cost of heapifying a node at height h is $O(h)$ and there are 2^{k-h} nodes at height h. Thus, the total cost of heapifying all nodes is

$$O\left(\sum_{h=1}^{k} h2^{k-h}\right) = O\left(2^k \sum_{h=1}^{k} d2^{-d}\right) = O(2^k) = O(n).$$

(SS9) There are n items in an array. It is easy to see that that their minimum can be found using $n-1$ comparisons and that $n-1$ are actually required. It is also easy to see that finding the max can similarly be done using $n-1$ comparisons with $n-1$ required.

Design an algorithm that finds *both* the minimum and the maximum using at most $\frac{3}{2}n + c$ comparisons, where $c > 0$ can be any constant you want.

Note: Although it is harder to prove, $\frac{3}{2}n + c$ comparisons is actually a lower bound.

Solution: *Assume n is even. Let the items be $x_0, x_1, \ldots, x_{n-1}$.*

- *Divide the items into $n/2$ pairs, $p_i = \{x_{2i}, x_{2i+1}\}$.*
- *Using $n/2$ comparisons compare the items in each p_i to each other, finding $m_i = \min\{x_{2i}, x_{2i+1}\}$ and $M_i = \max\{x_{2i}, x_{2i+1}\}$.*
- *Using $n/2 - 1$ comparisons find the minimum of the m_i:*
  using another $n/2 - 1$ comparisons find the maximum of the M_i.

  These are respectively, the min and max of the entire set and have been found using $3n/2 - 2$ comparisons.

If n is odd, keep one item x on the side and perform the algorithm above. Then use two further comparisons to compare x to the min and max found to compute the min and max of the entire set. This will use $3(n-1)/2 - 2 + 2 = 3n/2 - 3/2$ comparisons.

(SS10) You are given an input containing n/k lists:
(i) Each list has size k, and
(ii) for $i = 1$ to n/k, the elements in list $i-1$ are all less than all the elements in list i

The simple algorithm to fully sort these items is to sort each list separately and then concatenate the sorted lists together. This uses $\frac{n}{k}O(k \log k) = O(n \log k)$ comparisons.

Show that this is the best possible. That is, show that any comparison-based sorting algorithm to sort the n/k lists into one sorted list with n elements will need to make at least $\Omega(n \log k)$ comparisons.

Note that you can not derive this lower bound by simply combining the lower bounds for the individual lists.

Solution: *First calculate the number of different possible output (orderings).*

- *Each list has $k!$ possible orderings.*
- *Since all elements in list $i-1$ are less than the elements in list i, a final ordering can be any ordering of the first list, followed by any ordering of the 2nd list, etc..*

- $\Rightarrow$ *the* total *number of possible output orderings for the complete set of items is*

$$(k!)^{n/k}$$

- $\Rightarrow$ *the decision tree for sorting these n elements contains at least $(k!)^{n/k}$ leaves.*

Following the development of the lower bound for regular sorting in class, recall that a binary tree of height h has at most 2^h leaves. Thus, in our problem, height h of the comparison tree must satisfy

$$2^h \geq (k!)^{n/k} \quad \Rightarrow \quad h \geq \log((k!)^{n/k})$$

Note that, from Stirling's approximation,

$$\begin{aligned}\log((k!)^{n/k}) &= \frac{n}{k} \cdot \log(k!) \\ &= \frac{n}{k}\Theta(k \log k) \\ &= \Theta(n \log(k))\end{aligned}$$

Therefore, any comparison-based sorting algorithm requires $\Omega(n \log k)$ comparisons in the worst-case for solving this problem.

(SS11) **Prove that insertion in a binary search tree requires at least $\Omega(\log n)$ comparisons (in the worst case) per insertion, where n is the number of items in the search tree.**

Hint: What lower bounds have we learned in class? Suppose you built the search tree using insertions. What can you do with it?

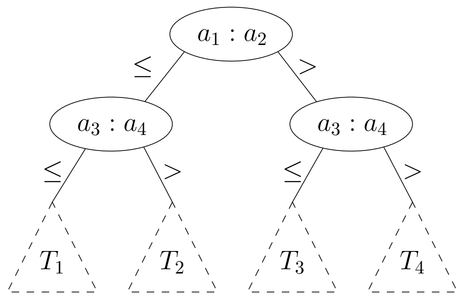
Solution: *After inserting into a binary search tree it takes only O(n) time to read the items out using an INORDER scan. This would produce the items in sorted order.*

This means that if you could insert in $o(logn)$ time per step (time being number of comparisons, $o(\log n)$ means asymptotically *faster than $\log n$) you could sort in $o(nlogn)$ time, which we proved in class was not possible.*

(SS12) **Build a Binary Search Tree for the items**
8, 4, 6, 13, 3, 9, 11, 2, 1, 12, 10, 5, 7
and draw the final tree.
Now, delete 3, 9, 4 in order and draw the resulting trees.

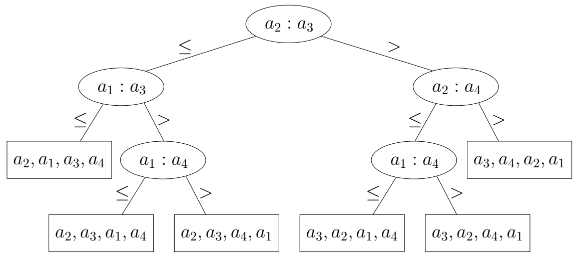
(SS13) The figure below shows part of the decision tree for mergesort operating on a list of 4 numbers, a_1, a_2, a_3, a_4.

Expand subtree T_3, i.e., show all the internal (comparison) nodes and leaves in subtree T_3.



Solution: *Mergesort on a_1, a_2, a_3, a_4 first starts by comparing a_1, a_2 and then comparing a_3, a_4 to get two sorted lists of size 2. It then merges those lists.*

In the problem, you are told that $a_2 < a_1$ and $a_3 \leq a_4$, which give you that the two sorted lists are "a_2a_1" and "a_3a_4". Tree T_3 below is the decision tree that merges those two sorted lists.



(SS14) The maximum item in a set of n real-valued keys is well defined. The maximum item in a set of n 2-dimensional real-valued points is not.

One definition that is used in database theory is that of *skyline vectors*. These are also known as *maximal points* or *maximal vectors*.

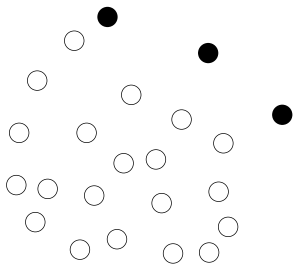
Let $S = \{p_1, p_2, \ldots, p_n\}$ be a set of 2-d points where $p_i = (x_i, y_i)$. A point $p \in S$ is a *skyline vector* if no other point is bigger than it in both x and y dimensions.

Formally p_j *dominates* p_i if

$$x_i < x_j \quad \text{and} \quad y_i < y_j.$$

$p = (x, y)$ *is a skyline vector in S* if no p_i in S dominates p.

In the example below, the 3 filled points are the skyline ones.



(a) Give an algorithm that finds the skyline vectors in a set S of n points in $O(n \log n)$ time.

(b) Suppose that the points all have integer coordinates in the range $[1, \ldots, n^2]$. Give an $O(n)$ algorithm for solving the same problem.

Solution:

To simplify we will assume that all the points have different x and y coordinates (repeated values require just a few more comparisons.).

(a) Sort the items from smallest to largest x coordinate in $O(n \log n)$ time and insert the items into an array in sorted order.

So the points are now $p_i = (x_i, y_i)$ with

$$x_1 \le x_2 \le \cdots \le x_n.$$

Note that with this ordering, p_i is maximal if and only if

$$y_i \ge \max_{j>i} y_j$$

This gives a simple algorithm.

i) Report that p_n is maximal. Set maxy $= y_n$.

ii) Walk from right to left in the array, i.e., looking at p_i with $i = n - 1$ to 1.
If $y_i >$ maxy, report that p_i is maximal and set maxy $= y_i$.

Total running time is $O(nlogn)$ for the sort $+O(n)$ for the scan through the array.

Note that this is not the only possible $O(n \log n)$ algorithm. There are many others. For example, it is possible to design a divide-and-conquer $O(n \log n)$ algorithm as well.

(b) replace the first sort by an $O(n)$ time radix sort (using base n for the numbers).

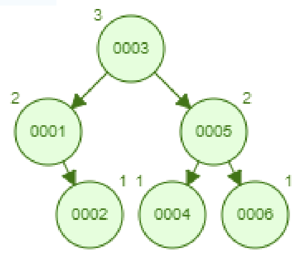
(SS15) **AVL Trees**

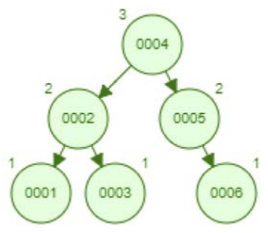
(a) Construct an AVL tree by inserting the items 134625 in that order.
Next construct another AVL tree on those items by inserting in the order 123456.
Do they have the same height?
Now construct an AVL tree by inserting the items 5362471 in that order and another by inserting 4261357 in that order. Do those two trees have the same height?

(b) What are the minimum and maximum heights for an AVL tree with 88 nodes labelled $1, 2, 3 \ldots, 88$?

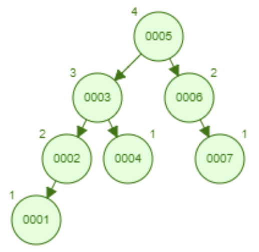
Solution:

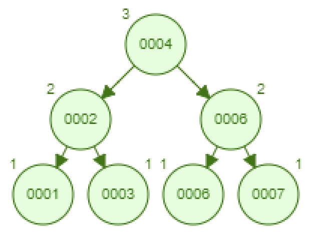
(a) The two trees below are the ones built for 134625 and then 123456. They do have the same height.





The two further trees below are the ones built for 5362471 and then 4261357. They do NOT have the same height.





You can construct these trees using the web site pointed to by the class lecture note page)

(b)
- *A binary tree of height h can hold at most $2^{h+1} - 1$ nodes (this is when it is full)*
- *The minimum height for ANY binary tree with n nodes is therefore*

$$h = \lceil \log_2 n + 1 \rceil - 1,$$

  which is the height of a complete tree with n nodes (all levels full except for possibly the bottom one).
- *Such a tree is an AVL tree as well, so this is the minimum for AVL trees too. In our case minimum $h = \lceil \log_2 89 \rceil - 1 = 6$.*
- *In class we learned that the minimum number of nodes in an AVL tree of height 8 is 88 so an AVL tree of height 8 with 88 nodes exists. The minimum number of nodes in an AVL tree of height 9 is $88 + 54 + 1 > 88$ and larger heights require even more nodes. So, the max height is 8.*

(SS16) **Sorting Polynomially Bounded Integers**

In this problem, you are given n integers to sort.

(a) You are told they are all in the range $[0, n^2 - 1]$. How fast can you sort them?

(b) Now you are told they are all in the range $[0, n^t - 1]$ for some fixed t. How fast can you sort them?

(SS17) **Lower Bound on the EPFL of Binary Trees**

The proof (in COMP3711H) that comparison-based sorting algorithms require $\Omega(n \log n)$ comparisons *on average* used the fact that the External Path Length of a binary tree with n leaves is at least $n \log_2 n + O(m)$. Prove this fact.

Solution:

In what follows let $\mathcal{T}_n$ be the tree that has minimal EPFL for n leaves. Let $T(n) = EPFL(\mathcal{T}_n)$ be the actual EPFL.

Claim 1: *Every internal node of $\mathcal{T}_n$ has 2 children.*

Proof: *Suppose this is incorrect. Then $\mathcal{T}_n$ contains an internal node v with exactly one child u. If v is the root of $\mathcal{T}_n$, then remove v and set u to be the root of the tree. Otherwise, let p denote the parent of v. Replace the pointer from p to v with a pointer from p to u.*

Both of these constructions keep the number of leaves the same but reduce the EPFL, contradicting the definition of $\mathcal{T}_n$.

Claim 2: *Let h be the depth of the lowest leaf in $\mathcal{T}_n$.*
Then all leaves of $\mathcal{T}_n$ are on level h or $h-1$

Proof: *Suppose not. Then there is some leaf u at depth $\ell \le h-2$. Let x be a leaf at depth h, v it's parent and y the other child of v. Claim 1 tells us that y exists*

Now create a new tree $\mathcal{T}'$ from $\mathcal{T}_n$ by removing x and y and making u an internal node with two children. Then $\mathcal{T}'$ also has n leaves and

$$EPFL(\mathcal{T}') = EPFL(\mathcal{T}_n) - (2h+\ell) + (h-1+2(\ell+1)) = EPFL(\mathcal{T}_n) - h + \ell + 1 \le EPFL(\mathcal{T}_n) - 1$$

contradicting the minimality of $\mathcal{T}_n$.

Claim 3: *Let h be the depth of the lowest leaf in $\mathcal{T}_n$.*
Then $\mathcal{T}_n$ contains 2^{h-1} nodes on level $h-1$ (those that are not leaves are internal nodes).

Proof: *First note that $\forall i \le h-2$, $\mathcal{T}_n$ contains 2^i internal nodes on level i. If not, let $i' \le h-2$ be the highest level for which this is incorrect. Then $\mathcal{T}_n$ contains a leaf on level i', contradicting claim 3.*

Since $\mathcal{T}_n$ contains 2^{h-2} internal nodes on level $h-2$ it contains 2^{h-1} nodes on level $h-1$.

Now let I be the number of internal nodes on level $h-1$. The number of leaves on level h is then $2I$ while the number of leaves on level $h-1$ is $2^{h-1} - I$.

Thus $n = 2I + (2^{h-1} - I) = 2^{h-1} + I$ where $1 \le I \le 2^{h-1}$. This implies $h = \lceil \log_2 n \rceil = \log_2 n + O(1)$.

Finally

$$EPFL(\mathcal{T}_n) = (h-1)(2^{h-1} - I) + 2hI = hn + O(n) = n \log_2 n + O(n).$$

(SS*1) Extra Problem. *The limits of comparison-based lower bounds*

The purpose of this problem is to illustrate that lower bounds in comparison-based models can completely fail in other models.

Let $S = \{x_1, x_2, \ldots, x_n\}$ be a set of integers or real numbers. Let $y_1, y_2, \ldots, y_n$ be the same numbers sorted in increasing order. The *MAX-GAP* of the original set is the value

$$\mathtt{Max}_{1 \le i < n}(y_{i+1} - y_i).$$

As an example, if $S = \{3, 12, 16, 7, 13, 1\}$, sorting the items gives $1, 3, 7, 12, 13, 16$ and the MAX-GAP is $12 - 7 = 5$.

Using a more advanced form of the $\Theta(n \log n)$ proof of the lower bound for sorting it can be proven that calculating MAX-GAP requires $\Theta(n \log n)$ operations if only comparisons and algebraic calculations are used. In this problem, we will see that, if the floor (truncate) operator $\lfloor x \rfloor$ can also be used, the problem can be solved using only $O(n)$ operations!

Before reading the solution below, try to see if you can solve it yourself!

- Find y_1 and y_n, the minimum and maximum values in S.
- Let $\Delta = \frac{y_n - y_1}{n-1}$. Let B_i be the half-closed half-open interval defined by
$$B_i = \Big[y_1 + (i-1)\Delta,\, y_1 + i\Delta\Big)$$
for $i = 1, 2, \ldots n-1$ and set $B_n = \{y_n\}$.
- Prune S as follows. For every B_i throw away all items in $S \cap B_i$ except for the smallest and largest. Let S' be the remaining set.
- Find the Max-Gap of S' by sorting S' and running through the items in S' in sorted order. Output this value

Prove that this algorithm outputs the correct answer and show that every step can be implemented in $O(n)$ time (the 3rd step might require the use of the floor function).

Solution:

First, we show correctness.

Note that $\sum_{i=1}^{n-1}(y_{i+1} - y_i) = y_n - y_1$.

We first note that

$$\max_{1 \le i < n} \frac{y_{i+1} - y_i}{n-1} \ge \Delta.$$

If not, we get the contradiction,

$$y_n - y_1 = \sum_{i=1}^{n-1}(y_{i+1} - y_i) \le (n-1)\left(\max_{1 \le i < n} \frac{y_{i+1} - y_i}{n-1}\right) < (n-1)\Delta = y_n - y_1.$$

If two points x, x' are in the same box B_k then $|x - x'| < \Delta$. So, the two points y_j, y_{j+1} forming the max gap must be in different boxes, with y_j the largest value in its box and y_{j+1} the smallest value in its box (with no points between them).

This means that step 4 of the algorithm will examine that particular value $y_{j+1} - y_j$ after sorting S'. To prove the correctness of the algorithm it only remains to show that non of the new gaps that occur in S' are larger *than $y_{j+1} - y_j \geq \Delta$.*

This is easy to see since the only new gaps that are created are between two points in the same box (after removing points between them) and any gaps between points in the same box are $< \Delta$.

Now we see show the $O(n)$ running time.

Step 1 can be implemented in $O(n)$ time using two linear scans; the first to find y_1 and the second to find y_n.

In linear time, calculate for each item x_i the key

$$k_i = \left\lfloor \frac{x_i - y_1}{\Delta} \right\rfloor + 1.$$

By definition $x_i \in B_{k_i}$. **Note that this is the only place in the algorithm we use the floor function.**

Now create two new Arrays of size n, MAX and MIN.
$MAX[k]$ will store the index i of the largest x_i with key k;
$MIN[k]$ will store the index of the smallest x_i with key k.
If there is no item with key k, set $MAX[k] = MIN[k] = 0$.

It is easy to see that these arrays can be filled in properly in $O(n)$ time.

Filling in these arrays actually implements Step 3, since we are throwing away all but the largest and smallest items in each box, i.e., finding S'.

S' can now be sorted in $O(n)$ time as follows. For i = 1 to N DO
If $MAX[k] = MIN[k] = 0$, don't write anything.
If $MAX[k] = MIN[k] \neq 0$, write $x_{MIN[k]}$.
If $MAX[k] \neq MIN[k]$ write $x_{MIN[k]}$, $x_{MAX[k]}$.

Since this writes down the 0,1,2 remaining items in each box in order and the items in each box are written before the items in following boxes, this writes down the items in S' in order.

Note: To reiterate, we started by stating that, if only comparisons and (algebraic) comparisons are allowed, MAX-GAP requires $\Omega(n \log n)$ time. But, we just showed how to solve MAX-GAP in linear time.

This is NOT a contradiction because the algorithm uses an additional operation, the truncation (floor) operation.

(SS*2) **AVL Trees**

Given any specific insertion order on n keys the output is a specific AVL tree. Recall that a tree T is an AVL tree if it satisfies the *local* balance condition at every node. This doesn't a-priori imply that every possible AVL tree can be constructed via insertions.

Suppose T is some tree on n keys that satisfies the AVL balance condition.

Is there always an insertion order that of the keys that builds T? If yes, show one; if no, show a counterexample.

Solution: *The answer is* **Yes there is**. *Order the keys by their depth in the tree, highest to lowest, breaking ties arbitrarily. For example, in the first AVL tree in problem 1, the ordering would be 134625. Build a tree by inserting the nodes into the tree in that order.*

The claim is that at every step, after doing the insertion, **the AVL condition is satisfied on the tree built so far, so no rebalancings are done.** *Thus, this is like inserting nodes into a STATIC (not AVL) tree in increasing depth order. Such an insertion order will build the original tree.*

The tricky part of this proof is to show that, if the original tree is an AVL tree then, after every insertion, no rebalancings need to be done. We will see this through the following intermediate lemma.

Lemma 1: Let T be an AVL tree with height h. Remove any subset of nodes at depth h. What remains is also an AVL tree.

Proof: The proof is by induction on h. The statement is true by observation for $h = 0, 1, 2$.

Now suppose the statement is true for all AVL trees of height $< h$.

Let T be an AVL tree of height h with root r and T' the same tree with a given subset of depth h nodes removed.

For u a node in T (and T') set

- *T_u to be the subtree in T rooted at u, (T'_u the subtree in T',)*
- *$h(u)$ the height of T_u (and $h'(u)$ the height of T'_u).*

Now

- *Let x be the left child of r and y the right child.*
  By definition, T_x, T_y are AVL trees with heights $< h$.
- *Removing a subset of nodes of depth h from T_x, T_y removes a set of (none, some or all) nodes from depth $h(x)$ in T_x and from depth $h(y)$ in T_y.*
- *Thus, by induction, T'_x and T'_y are AVL trees which means that all of their nodes satisfy the AVL balance condition.*

So, in order to prove the Lemma it only remains to show that node r satisfies the AVL balance condition in T'.

From the AVL condition, one of the following three statements must hold:
(i) $h(x) = h(y) = h - 1$ or (ii) $h(y) = h(x) - 1 = h - 2$ or (ii) $h(x) = h(y) = h - 2$.

(i) *after removing nodes at depth h, $h'(x)$ and $h'(y)$ are either h or $h-1$ so $|h'(x) - h'(y)| \leq 1$ and r satisfies the AVL condition.*

(ii) *In (ii) no nodes at depth h are in T_y so $h'(y) = h(y) = h-2$.*
On the other hand, $h(x) = h-1$.
The nodes at depth h in the original tree are at depth $h-1$ in $h'(x)$.
After removing (none, some or all) of those nodes to get T', we are left with $h'(x) = h-1$ OR $h'(x) = h-2$.
In both of those cases $|h'(x) - h'(y)| \leq 1$ and r satisfies the AVL condition.

(iii) *Symmetric to (ii)*

■

To apply the lemma return to the ordering of the keys defined at the top of this solution.

Let x be any key. Let h be the height of the tree and h' the depth of x.

Remove all the nodes on levels $h, h-1, h-2, \ldots, h'+1$ in that order. Multiple applications of the lemma tells us that what remains is an AVL tree.

Now remove x and all items after x in the ordering that are still in the tree (they must be on the same level h' as x).

Again the Lemma tells us that what remains will be an AVL tree.

Now insert x into the tree using the standard BST insertion algorithm. Because all of its ancestors are in the exact same location as they are in the original AVL tree, x is inserted into the same place as it was in the original AVL tree. The Lemma tells us that this tree is also an AVL tree with x and all of the items before it in the ordering in the same places.

Inserting every key into the tree using the defined ordering therefore reconstructs the original tree.

Randomization (R)

(R1) Consider the HIRE-ASSISTANT algorithm described in the lecture notes.

Assume that the candidates are presented in a random order.
The analysis in the lecture notes calculated the *Expected* number of hires. For this problem calculate:

(a) the probability that you hire exactly one person.

(b) the probability that you hire exactly n people.

Solution:

Recall that there are $n!$ different interview orderings that can be chosen and each ordering is equally likely to be chosen (with probability $1/n!$).

(a) Hiring exactly one time means that you hired the first person and that person was the best person. The probability of the first person being the best person is exactly $\frac{1}{n}$.

(b) Probability that you hire exactly n times means that the candidates are interviewed in increasing order of their quality. The probability of this particular unique random ordering being chosen is exactly $\frac{1}{n!}$

(R2) Use indicator random variables to solve the following, known as the **hat-check problem**.

Each of n customers gives a hat to a hat-check person at a restaurant.

The hat-check person gives the hats back to the customers in a random order.

What is the expected number of customers who get back their own hat?

Note: Replacing hats with homeworks and customers with students gives the following equivalent question: suppose that there are n students in a class who have just submitted their homework. The teacher gives the homeworks back to the students in a random order and asks the students to mark the homework they have been handed. What is the expected number of students who have been asked to mark their own homework?

Solution:

Set

$$X_i = \begin{cases} 1 & \textit{if the i-th customer get back his own hat} \\ 0 & \textit{otherwise} \end{cases}$$

Note that $E(X_i) = Pr(X_i = 1) = \frac{1}{n}$.

$$\begin{aligned} E(X) &= E\left(\sum_{i=1}^{n} X_i\right) \\ &= \sum_{i=1}^{n} E(X_i) = \sum_{i=1}^{n} \frac{1}{n} = 1 \end{aligned}$$

Note that the X_i's are not **independent.**
For example, when $X_1 = X_2 = ... = X_{n-1} = 1$, then X_n must be 1 as well.
But Independence is not needed for linearity of expectation.

(R3) **Another Analysis of Randomized Selection** We learned the *Randomized Selection* algorithm and used a geometric series analysis approach to show that it runs in $O(n)$ expected time.

In this problem you will rederive the $O(n)$ time in a different way, using the *Indicator Random Variable* method used to analyze Quicksort in the COMP3711 lectures.

Recall the *Randomized Selection* algorithm to find the k**-th smallest element**.

Pick a random pivot, divide the array into 3 parts:

left subarray, pivot item, right subarray.

We then either stop immediately or recursively solve the problem in the left **OR** the right part of the array.

- As in quicksort, denote the elements in **sorted order** by $z_1, \ldots, z_n$ (so we are searching for z_k)
- We use the same random model for choosing the pivot as for quicksort.

I) Define

$$X_{ij} = \begin{cases} 1 & \text{if } z_i \text{ and } z_j \text{ are compared by the algorithm} \\ 0 & \text{otherwise} \end{cases}$$

Prove the following three facts

(a) $i \le k \le j$: $\Pr[X_{ij} = 1] = 2/(j - i + 1)$.
(b) $i < j < k$: $\Pr[X_{ij} = 1] = 2/(k - i + 1)$.
(c) $k < i < j$: $\Pr[X_{ij} = 1] = 2/(j - k + 1)$.

(II) By the indicator random variable technique, the expected total number of comparisons is

$$\sum_{i<j} E[X_{ij}] = \sum_{i<j} \Pr[X_{ij} = 1]$$

Use this to show that $\sum_{i<j} E[X_{ij}] = O(n)$.

(R4) **Quicksort with repeated elements**

The discussion of the expected running time of randomized quicksort in the COMP3711 lecture notes assumed that all element values are distinct. In this problem, we examine what happens when they are not.

(a) Suppose that all element values are equal. What would randomized quicksort's running time be ?

(b) The PARTITION procedure taught returns an index q such that each element of $A[p...q-1]$ is less than or equal to $A[q]$ and each element of $A[q+1...r]$ is greater than $A[q]$.

Modify PARTITION to produce a new procedure PARTITION$'(A, p, r)$, which permutes the elements of $A[p...r]$ and returns two indices q, t, where $p \le q \le t \le r$, such that

- all elements of $A[q...t]$ are equal,
- each element of $A[p...q-1]$ is less than $A[q]$, and
- each element of $A[t+1...r]$ is greater than $A[q]$

Like PARTITION, your PARTITION$'$ procedure should take $\Theta(r-p)$ time.

(c) Modify the QUICKSORT procedure to produce QUICKSORT$'(A, p, r)$ that calls PARITITION$'$ but then only recurses on $A[p, q-1]$ and $A[t+1, r]$.

Problem (R*3) will discuss how to analyze this new algorithm

(R5) Let $A[1..n]$ be an array of n distinct numbers.
In class we said that if $i < j$ and $A[i] > A[j]$, then the pair (i, j) is called an **inversion** of A.

Suppose that the elements of A form a uniform random permutation of $\langle 1, 2, ..., n \rangle$.

Use indicator random variables to compute the expected number of inversions.

Solution:

For $i < j$ set

$$X_{i,j} = \begin{cases} 1 & \text{if } A[i] > A[j], \\ 0 & \text{otherwise.} \end{cases}$$

Now set

$$X = \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij}$$

to be the total number of inversions.

For any fixed i, j exactly half the set of $n!$ permutations have $A[i] > A[j]$ so

$$E(X_{ij}) = \Pr(X_{ij} = 1) = \frac{1}{2}.$$

Thus

$$\begin{aligned}
E(X) &= E\left(\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} X_{ij}\right) \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} E(X_{ij}) \\
&= \sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{1}{2} \\
&= \binom{n}{2}\frac{1}{2} = \frac{1}{2}\frac{n(n-1)}{2} = \frac{n(n-1)}{4}.
\end{aligned}$$

Both Bubble Sort and Insertion Sort perform exactly the Inversion Number *of swaps. So this problem is really showing that the expected number of swaps performed by both Bubble Sort and Insertion Sort on a "random input" of size n is $\frac{n(n-1)}{4}$.*

(R*1) Extra Problem. You are given n pairs of nuts and bolts

$$(N_1, B_1), (N_2, B_2), \dots, (N_n, B_n).$$

Each pair is a different size than the others. Someone has unscrewed all of the nuts off of the bolts and mixed them up.

Problem: Match all nuts up with their corresponding bolts.

If we could separately
(i) sort all the bolts by increasing size and then
(ii) sort all the nuts by increasing thread size
$\Rightarrow$ problem would be easily solvable in $O(n \log n)$ time.

After sorting, just match them up in order from smallest to largest. The difficulty is that we can't do this because we can't compare the sizes of two nuts directly or the sizes of two bolts directly.

The only operation available is to try to screw a bolt B into a nut N and then, by seeing whether the bolt

- (a) goes loosely in,
- (b) perfectly fits or
- (c) can't go in at all,

decide whether their thread sizes satisfy

$$\text{(a) } B < N, \quad \text{(b) } B = N \text{ or} \quad \text{(c) } B > n.$$

Design an $O(n \log n)$ time randomized algorithm for matching nuts and bolts.

Hint: Try to modify Quicksort.

Solution:

Let $X = \{B_1 \dots, B_n\}$, $Y = \{N_1 \dots, N_n\}$.

```
MATCH(X,Y)
1. If $|X| = |Y| = 1$ match the unique nut with the unique bolt.
   Otherwise
2.    Choose a nut $N$ at random from $Y$.
3.    Compare ALL bolts in $X$ with $N$; While doing this
        Let $B$ be unique bolt that fits $N$
        Split the remaining bolts into two sets
        $B_S$ = Bolts smaller than $N$,    $B_L$ = Bolts larger than $N$.
4.    Compare ALL nuts in $Y$ except for $N$ with $B$; Set
        $N_S$ = Nuts smaller than $B$,    $N_L$ = Nuts larger than $B$.
5.    Match $(B, N)$ and recursively call
6.       $MATCH(B_S, N_S)$,    $MATCH(B_L, N_L)$
```

Visualize X and Y as being stored in two separate ARRAYS.
Choosing the random nut in 2 is like choosing a random pivot upon which to partition

(the nuts) in Quicksort. Steps 3-4 can be implemented in $O(n)$ time using almost exactly the same code as the partition code in Quicksort.

After Steps 3,4,5 the X and Y arrays have been partitioned *around their pivots and the recursive calls are done into the two pairs of subarrays.*

The analysis is then EXACTLY like Quicksort and this algorithm takes $O(n \log n)$ Expected *time to match the nuts and bolts.*

The randomized algorithm from the previous pages was discovered quite early but it took a long time for researchers to find a good deterministic one. For a while, the best known deterministic algorithm ran in $O(n \log^4 n)$ time.

$O(n \log n)$ worst case time algorithms are now known to exist but, unlike in the case of sorting, these deterministic algorithms are MUCH more complicated than the randomized one.

For more information about this problem, Google the phrase Matching Nuts and Bolts Algorithms.

(R*2) Extra Problem. **Randomized Binary Search Trees**

- Consider a Binary search tree T on n keys.
  The *depth*, $d(v)$, of v in T is the length of the path from the root of T to v. Note that the depth of the root is 0. The *Path Length of* T, $PL(T)$, is the sum of the depths of all of the nodes of T; $PL(T) = \sum_{v \in T} d(v)$.

  Note that $\frac{1}{n} PL(T)$ is the average depth of a node in the tree. This is also the average time to search for a randomly chosen node in the tree.
- Suppose that every key K_i in a set of n keys has real weight w_i associated with it, with the weights being unique.
- There is a unique binary search tree that can be built on the n keys that also satisfies min-heap order by the weights (Why?).
- Suppose n weights $w_1, w_2, \ldots, w_n$ are chosen independently at random from the unit interval $[0, 1]$ and then sorted. The resulting order is a random permutation of the n items.

A *Treap* or *Randomized Binary Search Tree* on n keys K_i is constructed by choosing n weights w_i independently at random from the unit interval $[0, 1]$ and associating w_i with K_i. The Treap is the unique BST built on the n keys that also satisfies min-heap order on the weights.

(a) Describe how to build T in time $O(n \log n + PL(T))$

Solution: *After picking the random weights sort the keys by their weights, from largest to smallest. Then insert the keys into the tree in that largest to smallest weight order. Notice that this exactly builds the associated Treap (this can be proven by induction) and the cost of inserting a node is exactly its depth. Thus, the cost of building the tree is $O(n \log n)$ for the sort plus $PL(T)$, which is what we wanted to prove.*

(b) If T is the Treap built, prove that the average value of $PL(T)$ is $O(n \log n)$
Hint: consider Quicksort

Solution: Choosing the random weights fixes a random permutation.
We will show two different derivations. The first is via a recurrence relation similar to that of Quicksort's. The second is via Indicator Random Variables.

Solution 1:

Note the following fact. Let $\pi = \pi_1, \pi_2, \ldots, \pi_n$ be a random permutation on n items. Let π^1 be the items with value less than π_1 writen in the same order as in π and let π^2 be the items with value greater than π_1 writen in the same order as in π. Then π^1 is a random permutation on its items (i.e., each one of its $|\pi^1|!$ orderings is equally likely to occur) and π^2 is a random permutation on its items.
The process described essentially builds the tree from a permutation π as follows:

- *Choose the first item in the permutation as the root of the tree.*
- *If π^1 is not empty, recursively build the tree off of π^1 and make it the left subtree of π_1.*

- *If π^2 is not empty, recursively build the tree off of π^2 and make it the right subtree of π_1.*

Let C_n be the average value of $PL(T)$ where T is built from a random π on n items. From the construction described above, if π_1 is the kth key, then the left subtree is a random Treap on $k-1$ items so it has average path length C_{k-1} and the right subtree is a random Treap on the $n-k$ items to the right so it has average path length C_{n-k}. Since every item in the final tree is one level deeper than it is in the left or right subtree we find that the average path length, CONDITIONED ON THE ROOT being the kth key is

$$n-1+C_{k-1}+C_{n-k}.$$

Since every one of the n keys is equally likely to be the root

$$C_n = \frac{1}{n}\sum_{i=1}^{n}(n-1+C_{k-1}+C_{n-k}) = n-1+\frac{1}{n}\sum_{i=1}^{n}(C_{k-1}+C_{n-k})$$

with initial condition $c_0 = 0$. This is EXACTLY the quicksort recurrence, so $c_n = O(n\log n)$.

Solution 2:

Let $a_1, a_2, \ldots, a_n$ be the n keys in sorted order. The goal is to find

$$E(PL(T)) = E\left(\sum_{i=1}^{n} d(a_i)\right) = \sum_{i=1}^{n} E(d(a_i)).$$

Set

$$X_{i,j} = \begin{cases} 1 & \textit{if } a_i \textit{ is an ancestor of } a_j \textit{ in the Treap,} \\ 0 & \textit{otherwise} \end{cases}$$

(Recall that a is an ancestor of b if the path from the root to b passes through a.) Since the depth of a node is the number of its ancestors.

$$d(a_i) = \sum_{i\neq j} X_{i,j}$$

and

$$E(d(a_i)) = E\left(\sum_{i\neq j} X_{i,j}\right) = \sum_{i\neq j} E(X_{i,j}).$$

By the properties of Indicator Random Variables

$$E(X_{i,j}) = \Pr(X_{i,j}=1) = \Pr(a_i \textit{ is an ancestor of } a_j).$$

Let I be the interval between i and j. This is $[i,\ldots,j]$ if $i<j$ and$[j,\ldots,i]$ $j<i$. Note that the number of items in I is $|I| = |j-i+1|$.
Consider now how a_i could be an ancestor of a_j in the Treap.

Recall that the a_i are permuted by random permutation π so that a_i is placed in position π_i in the permuted order. From part (a) the Treap is built by inserting the a_i one at a time in that permuted order.

Let $k \in I$ be the index of the first item in I that appears in π. a_k will then be the first item with an index in I to be placed in the Treap. Note that every other item in I will be BELOW a_k in the tree (why?) and every other item in I will be compared to a_k on its way down the tree. There are three options

- *$k = j$: This is not possible since this would make a_j an ancestor of a_i.*
- *$k \in I \setminus \{i, j\}$: this is not possible because both a_i and a_j will be compared to a_k when walking down the tree and one will be placed to the left of a_k and the other to the right of a_k so neither a_i nor a_j could be an ancestor of the other.*
- *$k = i$: In this case a_i is an ancestor of a_j.*

Then

$$\begin{aligned}\Pr(a_i \textit{ is an ancestor of } a_j) &= \Pr(i \textit{ is the first item of } I \textit{ to appear in } \pi)\\ &= \frac{1}{|I|} = \frac{1}{|j-i|+1}.\end{aligned}$$

Thus

$$\begin{aligned}E(PL(T)) &= \sum_{i=1}^{n} E(d(a_i))\\ &= \sum_{i=1}^{n}\left(\sum_{j\neq i} E(X_{i,j})\right)\\ &= \sum_{i=1}^{n}\left(\sum_{j\neq i} \frac{1}{|j-i+1|}\right)\\ &= \sum_{1\le i<j\le n} \frac{2}{j-i+1}\\ &= 2\sum_{i=1}^{n-1}\sum_{j=i+1}^{n} \frac{1}{j-i+1}\\ &= 2\sum_{i=1}^{n-1}\sum_{t=2}^{n-i+1} \frac{1}{t}\\ &= 2\sum_{i=1}^{n-1}(H_{n-i+1}-1) = 2\sum_{t=2}^{n}(H_t - 1)\\ &= 2\left(\sum_{t=2}^{n} H_t\right) - 2(n-1)\\ &= 2\left(\sum_{t=2}^{n} \Theta(\log t)\right) - 2(n-1)\\ &= 2\Theta(\log n!) - 2(n-1)\\ &= \Theta(n\log n)\end{aligned}$$

Going Further: The Treap data structure, as described, is for a **Static** *dictionary, i.e., all keys/weights are assumed to be known in advance of construction.*

It is not difficult to extend this to be a **dynamic** *data structure, i.e., allowing key insertion and deletion. It can be proven that the expected insertion and deletion costs are $O(\log n)$ as well.*

For insertion, a new key is assigned a new random weight and is then inserted into the Treap using the standard BST insertion algorithm.

After insertion the new node is rotated up, using tree rotations similar to those in the AVL rebalancing, so that the Treap satisfies the min-heap condition. It can be shown that this requires $O(\log n)$ average case rotations so insertion is $O(\log n)$ on average.

Deletion uses similar ideas.

(R*3) **Analyzing Modified Quicksort from Problem (R4)**

Our analysis of QUICKSORT in class assumed that all elements were distinct.

Problem (R4) developed a more sophisticated version of Quicksort that handles repeated items more efficiently.

How would you adjust the analysis (binary tree plus indicator random variables) in the lecture notes to avoid the assumption that all elements are distinct and prove that QUICKSORT$'$ runs in $O(n)$ time.

(R*4) **Recursive Independence in the Probabilistic Analysis of Quicksort**

The probabilistic analysis of Quicksort (in COMP3711H) used the fact that if $\mathbf{A}$ is the set of items in subarray $A[p \ldots r]$ and π is a random permutation of $\mathbf{A}$ then

(a) $A[r]$ is equally likely to be any item in $\mathbf{A}$

(b) After running the partition algorithm on $A[p \ldots r]$, the input to the new left and right subproblems are again random permutations.

Prove the correctness of this fact.

Solution: *Without loss of generality we will assume that $p = 1$ and $r = n$.*

The analysis will use standard permutation notation in which S_n denotes the set of all $n!$ permutations on $\{1 \ldots n\}$.

Let $\pi \in S_n$. We write $\pi = \pi(i), \pi(2), \ldots, \pi(n)$. In particular, saying that π is a permutation of A means that after setting the permutation, $A[i] = \pi(i)$.

Recall that in this context, $\pi \in S_n$ being a "random" permutation means that each of the $n!$ possible permutations occurs with probability $1/n!$.

Let k be fixed. $\mathbf{1}_k \in S_k$ will denote the identity permutation on $\{1, \ldots, k\}$, i.e., $\mathbf{1}_k(i) = i$. Let $\alpha, \beta \in S_k$; $\alpha \circ \beta$ will denote the composition permutation satisfying $(\alpha \circ \beta)(i) = \alpha(\beta(i))$. α^{-1} is the inverse permutation. i.e., $\alpha \circ \alpha^{-1} = \alpha^{-1} \circ \alpha = \mathbf{1}_k$.

(a) For each r, there are $(n-1)!$ permutations of $\{1,\dots,n\}\setminus\{r\}$ so there are $(n-1)!$ permutations of $\{1,\dots n\}$ with $A[n]=r$. Thus,

$$\Pr(A[n]=r)=\frac{(n-1)!}{n!}=\frac{1}{n}.$$

(b) Define

$$S_{n,r}=\{\pi\in S_n\ :,\pi(n)=r\}$$

and

$$S'_{n,r}=\{\tau\in S_n\ :\ \forall i<r,\ \tau(i)<r,\quad \textit{and}\quad \tau(r)=r,\quad \textit{and}\quad \forall i>r,\ \tau(i)>r\}.$$

PARTITION can be viewed as a function that starts with input $\pi\in S_{n,r}$ and outputs $\tau\in S'_{n,r}$. This will be denoted as $\tau=P(\pi)$.

Note that $|S_{n,r}|=(n-1)!$ and $|S'_{n,r}|=(r-1)!(n-r)!$.

To proceed, we introduce alternative unique ways of writing permutations in $S_{n,r}$ and $S'_{n,r}$.

$\underline{\pi(r,I,\sigma_1,\sigma_2)\ :\ \textit{For}\ \pi\in S_{n,r}.}$

Let $I=\subseteq\{1\dots n-1\}$ be a set of $r-1$ indices, i.e, $|I|=r-1$. Set $I'=\{1,\dots,n-1\}\setminus I$.

Label I's indices as $a_1<a_2<\cdots<a_{r-1}$ and I''s indices as $a'_1<a'_2<\cdots<a'_{n-r}$.

Let $\sigma_1\in S_{r-1}$ and $\sigma_2\in S_{n-r}$. $\pi(r,I,\sigma_1,\sigma_2)$ denotes the unique *permutation satisfying*

$$A[n]=r,\quad \forall 1\le i<r,\ A[a_i]=\sigma_1(i),\quad \forall 1<i\le n-r,\ A[a'_i]=r+\sigma_2(i).$$

Note that every $\pi\in S_r$ can be written UNIQUELY in the form $\pi(r,I,\sigma_1,\sigma_2)$.

$\underline{\tau(r,\sigma'_1,\sigma'_2)\ :\ \textit{For}\ \tau\in S'_{n,r}.}$

For fixed r, let $\sigma'_1\in S_{r-1}$ and $\sigma'_2\in S_{n-r}$.
Let $\tau(r,\sigma'_1,\sigma'_2)$ denote the unique permutation such that

$$A[r]=r,\quad \forall 1\le i<r,\ A[i]=\sigma'_1(i),\quad \forall 1\le i\le n-r,\ A[r+i]=r+\sigma'_2(i).$$

Given this notation, an equivalent more formal statement of (b) is

(b') Fix r. If π is chosen uniformly at random from $S_{n,r}$ then

$$\forall\sigma'_1\in S_{r-1},\quad \Pr\Big(P(\pi)=\tau(r,\sigma'_1,\sigma'_2)\ \textit{for some}\ \sigma'_2\in S_{n-r}\Big)\ =\ \frac{1}{(r-1)!},\tag{1}$$

$$\forall\sigma'_2\in S_{n-r},\quad \Pr\Big(P(\pi)=\tau(r,\sigma'_1,\sigma'_2)\ \textit{for some}\ \sigma'_1\in S_{r-1}\Big)\ =\ \frac{1}{(n-r)!}.\tag{2}$$

It now remains to prove Equations (1) and (2).

The main observation is that the PARTITION algorithm only works by classifying items as being in the set $\{1,\dots,r-1\}$ or $\{r+1,\dots,n\}$. It never actually looks at their values. A closer look then shows that

(i) **For all** I, $\sigma_1 \in S_{r-1}$**, and** $\sigma_2 \in S_{n-r}$,
$P(\pi(r, I, \sigma_1, \sigma_2)) = \tau(r, \sigma_1, \sigma_2')$ **for some** $\sigma_2' \in S_{n-r}$**.**
More specifically, all of the items in $\{1, \ldots, r\}$ *stay in the same order in the output. This has no dependence on* I *or* σ_2*.*

(ii) Let s_I *denote the unique permutation such that*

$$P(\pi(r, I, \sigma_1, \mathbf{1}_{n-r})) = \tau(r, \sigma_1, s_I).$$

Note that this does NOT depend upon σ_1*.*
Then for every permutation $\alpha \in S_{n-r}$,

$$P(\pi(r, I, \sigma_1, \alpha)) = \tau(r, \sigma_1, \alpha \circ s_I).$$

(iii) Fix I*. Let* $\sigma_1 \in S_{r-1}$ *and* $\sigma_2 \in S_{n-r}$*. Then (i) and (ii) imply that* $\beta = \sigma_2 \circ (s_I)^{-1}$ *is the unique permutation in* S_{n-r} *such that*

$$\tau(r, \sigma_1, \sigma_2) = P(\pi(r, I, \sigma_1, \beta).$$

Since (iii) is true for every I *and there are* $\binom{n-1}{r-1}$ *possible choices of* I *this implies that*

$$\forall \tau \in S'_{n,r}, \quad |P^{-1}(\tau)| = \frac{1}{\binom{n-1}{r-1}} = \frac{(n-1)!}{(r-1)!(n-r)!} = \frac{|S_{n,r}|}{|S'_{n,r}|}.$$

($P^{-1}(\tau)$ *is the preimage of* τ *under PARTITION.)*
Thus if π *is chosen uniformly at random from* $S_{n,r}$ *then*

$$\forall \sigma_1' \in S_{r-1},\, \sigma_2' \in S_{n-r} \quad \Pr\Big(P(\pi) = \tau(r, \sigma_1', \sigma_2')\Big) = \frac{|P^{-1}(\tau(r, \sigma_1', \sigma_2')|}{|S_{n,r}|} = \frac{1}{|S_{n,r}|} = \frac{1}{(r-1)!(n-r)!}.$$

Thus Equation (1) follows from

$$\begin{aligned}\forall \sigma_1' \in S_{r-1}, \quad \Pr\Big(P(\pi) = \tau(r, \sigma_1', \sigma_2') \textit{ for some } \sigma_2' \in S_{n-r}\Big) &= \sum_{\sigma_2' \in S_{n-r}} \Pr\Big(P(\pi) = \tau(r, \sigma_1', \sigma_2')\Big) \\ &- (n-r)!\frac{1}{(r-1)!(n-r)!} \\ &= \frac{1}{(r-1)!}.\end{aligned}$$

Similarly Equation (2) follows from

$$\begin{aligned}\forall \sigma_2' \in S_{n-r}, \quad \Pr\Big(P(\pi) = \tau(r, \sigma_1', \sigma_2') \textit{ for some } \sigma_1' \in S_{r-1}\Big) &= \sum_{\sigma_1' \in S_{r-1}} \Pr\Big(P(\pi) = \tau(r, \sigma_1', \sigma_2')\Big) \\ &- (r-1)!\frac{1}{(r-1)!(n-r)!} \\ &= \frac{1}{(n-r)!}.\end{aligned}$$

Divide & Conquer (DC)

(DC1) Derive asymptotic upper bounds for $T(n)$. Make your bounds as tight as possible.

(a) You may assume n is a power of 2.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n/2) + n \qquad \text{if } n > 1 \end{aligned}$$

(b)

$$\begin{aligned} T(1) &= T(2) = 1 \\ T(n) &= T(n-2) + 1 \qquad \text{if } n > 2 \end{aligned}$$

(c) You may assume n is a power of 3.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n/3) + n \qquad \text{if } n > 1 \end{aligned}$$

(d) You may assume n is a power of 2.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 4T(n/2) + n \qquad \text{if } n > 1 \end{aligned}$$

(e) You may assume n is a power of 2.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T(n/2) + n^2 \qquad \text{if } n > 2 \end{aligned}$$

(f) You may assume n is a power of 2.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= T(n/2) + \log_2 n \qquad \text{if } n > 1 \end{aligned}$$

(g) You may assume n is a power of 2.

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 2T(n/2) + \log_2 n \qquad \text{if } n > 1 \end{aligned}$$

Solution:

(a) Set $h = \log_2 n$

$$\begin{aligned}
T(n) &= n + T\left(\frac{n}{2}\right) \\
&= n + \frac{n}{2} + T\left(\frac{n}{2^2}\right) \\
&= n + \frac{n}{2} + \frac{n}{2^2} + T\left(\frac{n}{2^3}\right) \\
\cdots \\
&= n + \frac{n}{2} + \frac{n}{2^2} + \cdots + \frac{n}{2^{h-2}} + \frac{n}{2^{h-1}} + T\left(\frac{n}{2^h}\right) \\
&= n\left(1 + \frac{1}{2} + \frac{1}{2^2} + \cdots + \frac{1}{2^{h-2}} + \frac{1}{2^{h-1}}\right) + T\left(\frac{n}{2^h}\right) \\
&= n\Big(2\left(1 - 2^{-h}\right)\Big) + T\left(1\right) \\
&\leq 2 \cdot n + 1 \\
T(n) &= O(n)
\end{aligned}$$

(b)

$$\begin{aligned}
T(n) &= T(n-2) + 1 \\
&= T(n - 2 \cdot 2) + 2 \\
&= T(n - 3 \cdot 2) + 3 \\
\cdots \\
&= T\left(n - \left\lfloor \frac{n-1}{2} \right\rfloor \cdot 2\right) + \left\lfloor \frac{n-1}{2} \right\rfloor
\end{aligned}$$

Since $n - 2\left\lfloor \frac{n-1}{2} \right\rfloor = 0, 1$ *depending upon whether* n *is odd, even and* $T(1) = T(2) = 1$ *we get*

$$T(n) = 1 + \left\lfloor \frac{n-1}{2} \right\rfloor = \lceil (n/2) \rceil = O(n)$$

(c) Set $h = \log_3 n$

$$\begin{array}{rcl}
T(n) & = & n + T(n/3) \\
& = & n + n/3 + T(n/3^2) \\
& = & n + n/3 + n/3^2 + T(n/3^3) \\
\dots & & \\
& = & n + n/3 + n/3^2 + \cdots + n/3^{h-2} + n/3^{t-1} + T(n/3^t) \\
\dots & & \\
& = & n + n/3 + n/3^2 + \cdots + n/3^{h-2} + n/3^{h-1} + T(n/3^h) \\
& = & n(1 + 1/3 + 1/3^2 + \cdots + 1/3^{h-2} + 1/3^{h-1}) + T(n/3^h) \\
& \leq & n\sum_{i=0}^{\infty}\left(\frac{1}{3}\right)^i + T(n/3^h) \\
& = & 3n/2 + T(1) \\
\Rightarrow T(n) & = & O(n)
\end{array}$$

(d) Set $h = \log_2 n$

$$\begin{array}{rcl}
T(n) & = & n + 4T\left(\frac{n}{2}\right) \\
& = & n + 4\Big(\frac{n}{2} + 4T\left(\frac{n}{2^2}\right)\Big) \\
& = & n + 2n + 4^2T\left(\frac{n}{2^2}\right) \\
& = & n + 2n + 2^2n + 4^3T\left(\frac{n}{2^3}\right) \\
\dots & & \\
& = & n + 2n + 2^2n + \dots 2^{t-1}n + 4^tT\left(\frac{n}{2^t}\right) \\
\dots & & \\
& = & n + 2n + 2^2n + \dots 2^{h-1}n + 4^hT\left(\frac{n}{2^h}\right) \\
& = & n(2^h - 1) + 4^h \\
\Rightarrow T(n) & = & n(n-1) + 4^h = O(n^2)
\end{array}$$

(e) *Set* $h = \log_2 n$

$$\begin{aligned}
T(n) &= n^2 + 3T(n/2) \\
&= n^2 + 3\left(\left(\frac{n}{2}\right)^2 + 3T\left(\frac{n}{2^2}\right)\right) \\
&= n^2 + \frac{3}{4}n^2 + 3^2T\left(\frac{n}{2^2}\right) \\
&= n^2 + \frac{3}{4}n^2 + 3^2\left(\left(\frac{n}{2^2}\right)^2 + 3T\left(\frac{n}{2^3}\right)\right) \\
&\dots \\
&= n^2 + \frac{3}{4}n^2 + \left(\frac{3}{4}\right)^2 n^2 + \ldots + \left(\frac{3}{4}\right)^{t-1} n^2 + 3^tT\left(\frac{n}{2^t}\right) \\
&\dots \\
&= n^2 + \frac{3}{4}n^2 + \left(\frac{3}{4}\right)^2 n^2 + \ldots + \left(\frac{3}{4}\right)^{h-1} n^2 + 3^hT\left(\frac{n}{2^h}\right) \\
&= n^2\sum_{i=0}^{h-1}(3/4)^i + 3^h
\end{aligned}$$

Note that $\sum_{i=0}^{h-1}(3/4)^i \leq \sum_{i=0}^{\infty}(3/4)^i$ *which converges, so* $\sum_{i=0}^{h-1}(3/4)^i = O(1)$ *(i.e., it's bounded independent of* n*). Also note that*

$$3^h = \left(2^{\log_2 3}\right)^{\log_2 n} = \left(2^{\log_2 n}\right)^{\log_2 3} = n^{\log_2 3} = O(n^2).$$

Thus

$$T(n) \leq n^2\sum_{i=0}^{h-1}(3/4)^i + 3^h = O(n^2).$$

(f) *Set* $h = \log_2 n$.

$$\begin{aligned}
T(n) &= \log_2 n + T\left(\frac{n}{2}\right) \\
&= \log_2 n + \log_2\frac{n}{2} + T\left(\frac{n}{2^2}\right) \\
&\dots \\
&= \log_2 n + \log_2\frac{n}{2} + \ldots + \log_2\frac{n}{2^{t-1}} + T\left(\frac{n}{2^t}\right) \\
&\dots \\
&= \log_2 n + \log_2\frac{n}{2} + \ldots + \log_2\frac{n}{2^{h-1}} + T\left(\frac{n}{2^h}\right) \\
&= T(1) + \sum_{i=0}^{h-1}\log_2\frac{n}{2^i} = 1 + \sum_{i=0}^{h-1}(\log_2 n - i) \\
&= 1 + h\log_2 n - h(h-1)/2 = 1 + h^2 - h(h-1)/2 \\
&= O(h^2) = O(\log^2 n)
\end{aligned}$$

(g) Set $h = \log_2 n$

$$\begin{aligned}
T(n) &= \log_2 n + 2T\left(\frac{n}{2}\right) \\
&= \log_2 n + 2\left(\log_2 \frac{n}{2} + T\left(\frac{n}{2^2}\right)\right) \\
&= \log_2 n + 2\log_2 \frac{n}{2} + 2^2 T\left(\frac{n}{2^2}\right) \\
\cdots & \\
&= \log_2 n + 2\log_2 \frac{n}{2} + 2^2 \log_2 \frac{n}{2^2} + \cdots + 2^{h-2}\log_2 \frac{n}{2^{h-2}} + 2^{h-1}\log_2 \frac{n}{2^{h-1}} + 2^h T\left(\frac{n}{2^h}\right) \\
&= \sum_{i=0}^{h-1}\left(2^i(\log_2 n - i)\right) + 2^h \\
&= n + \sum_{i=0}^{h-1}\left(2^i(h-i)\right) \\
&= n + 2^h \sum_{i=0}^{h-1}\left(2^{i-h}(h-i)\right) \\
&= n + 2^h \sum_{j=1}^{h}\left(j2^{-j}\right) \quad \textit{(change of variable } j = h - i\textit{)} \\
&\leq n + n\sum_{j=1}^{\infty}\left(j2^{-j}\right) \\
&= O(n)
\end{aligned}$$

where the last inequality comes from the fact that $\sum_{j=1}^{\infty}\left(j2^{-j}\right) < \infty$.

(DC2) Using the *Master Theorem*, give asymptotic tight bounds for $T(n)$

(a)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T(n/4) + n \qquad \text{if } n > 1 \end{aligned}$$

(b)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 3T(n/4) + 1 \qquad \text{if } n > 1 \end{aligned}$$

(c)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 4T(n/2) + n^2 \qquad \text{if } n > 1 \end{aligned}$$

(d)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 4T(n/3) + n^2 \qquad \text{if } n > 1 \end{aligned}$$

(e)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 9T(n/3) + n^2 \qquad \text{if } n > 1 \end{aligned}$$

(f)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 10T(n/3) + n^3 \qquad \text{if } n > 1 \end{aligned}$$

(g)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 99T(n/10) + n^2 \qquad \text{if } n > 1 \end{aligned}$$

(h)

$$\begin{aligned} T(1) &= 1 \\ T(n) &= 101T(n/10) + n^2 \qquad \text{if } n > 1 \end{aligned}$$

Solution:

Recall the version of the Master Theorem we saw.

Let $a, b \geq 1$ and $c \geq 0$ be constants.

The recurrence

$$T(n) = aT(n/b) + n^c$$

has the following solution (independent of initial conditions):

Case 1 $c < \log_b a$: *Then* $T(n) = \Theta\left(n^{\log_b a}\right)$

Case 2 $c = \log_b a$: *Then* $T(n) = \Theta\left(n^c \log n\right)$

Case 3 $c > \log_b a$: *Then* $T(n) = \Theta\left(n^c\right)$

(a) *This is* case 3 *because* $1 > \log_4 3$, *so*

$$\mathbf{T(n) = \Theta(n)}.$$

(b) *This is* case 1 *because* $0 < \log_4 3$, *so*

$$\mathbf{T(n) = \Theta(n^{\log_4 3}) = \Theta(n^{0.7924\ldots})}.$$

(c) *This is* case 2 *because* $2 = \log_2 4$, *so*

$$\mathbf{T(n) = \Theta(n^2 \log n)}.$$

(d) *This* case 3 *because* $2 > \log_3 4$, *so*

$$\mathbf{T(n) = \Theta(n^2)}.$$

(e) *This is* case 2 *because* $2 = \log_3 9$, *so*

$$\mathbf{T(n) = \Theta(n^2 \log n)}.$$

(f) *This is* case 3 *because* $3 > \log_3 10$, *so*

$$\mathbf{T(n) = \Theta(n^3)}.$$

(g) *This is* case 3 *because* $2 > \log_{10} 99$, *so*

$$\mathbf{T(n) = \Theta(n^2)}.$$

(h) *This* case 1 *because* $2 < \log_{10} 101$, *so*

$$\mathbf{T(n) = \Theta(n^{\log_{10} 101})}.$$

(DC3) **Using Black-box median algorithms (modified from CLRS)**
For this problem, you assume that you are given a black-box $O(n)$ time algorithm for finding the median ($\lceil n/2 \rceil$nd) item in a size n array. This means that you can call the algorithm and use its result but can't peer inside of it.

(a) Show how *Quicksort* can be modified to run in $O(n \log n)$ *worst case* time.

(b) Give a simple linear-time algorithm that solves the selection problem for an arbitrary order statistic. That is, given k, your algorithm should find the k'th smallest item.

(c) For n distinct elements $x_1, x_2, \ldots, x_n$ with associated positive weights $w_1, w_2, \ldots, w_n$ such that $\sum_{i=1}^{n} w_i = 1$, the **weighted (lower) medium** is the element x_k satisfying

$$\sum_{x_i < x_k} w_i < \frac{1}{2} \quad \text{and} \quad \sum_{x_i > x_k} w_i \le \frac{1}{2}.$$

If the x_i are sorted, then it is easy to solve this problem in $O(n)$ time by just summing up the weights from left to right and walking through the sums until k is found. Show that if the items are *not* sorted you can still solve the problem in linear time using the black box median finding algorithm.

Solution:

(a) Before every partition step in Quicksort use the $O(n)$ black box algorithm to find the median of the current subarray. Then use that median as the pivot. This splits the subarray into two (almost) equal parts so the running time of the full algorithm satisfies

$$T(n) = 2T(n/2) + O(n)$$

implying $T(n) = O(n \log n)$.

(b) Just modify the randomized selection algorithm so that,at every step, instead of choosing the pivot at random from the current subarray. it uses the $O(n)$ median finding algorithm to find the median and then uses the median as pivot. At each step, the algorithm will reduce the size of the array in which it is searching by $1/2$ so the running time satsifies

$$T(n) \le T(n/2) + O(n)$$

which yields $T(n) = O(n)$.

(c) If there are 2 items we can solve the problem in $O(1)$ time.

Otherwise, the idea is to "simulate" the binary search without actually sorting by using the selection algorithm and partitioning.

In $O(n)$ time find the median x_m of the x_j. In another $O(n)$ time calculate $W_L = \sum_{x_j \le x_m} w_j$. Note that $W_R = \sum_{x_j < x_m} w_j = 1 - W_L$.

If $W_R \le 1/2$ then we know that $x_k \le x_m$.

Then throw away all the (point,weight) pairs with $x_i > x_m$ and add W_R to w_m. Note that, for the new (remaining) weights, $\sum w_i = 1$ and the weighted median of this *smaller set of weighted points is the same as the weighted median of the original set. Recurse to find the weighted median of the new set.*

If $W_R > 1/2$ then we know that $w_k > w_m$.
Then throw away all the (point,weight) pairs with $x_i < x_m$ and set $w_m = W_L$. Note that, for the new (remaining) weights, $\sum w_i = 1$ and the weighted median of this smaller set of weighted points is the same as the weighted median of the original set. Recurse to find the weighted median of the new set.

Each recursion splits the problem by 1/2 so

$$T(n) = T(n/2) + O(n)$$

and $T(n) = O(n)$.

(DC4) **Polynomial Evaluation** The input to this problem is a set of $n+1$ coefficients $a_0, a_1, \ldots, a_n$. Define $A(x) = \sum_{i=0}^{n} a_i x_i$

(a) Given value x, how can you evaluate $A(x)$ using $O(n)$ multiplications and $O(n)$ additions?
Can you evaluate $A(x)$ using at most n multiplications and n additions?

(b) Now suppose that $A(x)$ has at most k non-zero terms. How can you evaluate $A(x)$ using $O(k \log n)$ operations.
Hint. How can you evaluate x^n using $O(\log n)$ operations.

Solution:

(a) Note that

$$\sum_{i=0}^{n} a_j x_j = a_0 + x(a_1 + x(a_2 + x(\cdots(a_{n-2} + x(a_{n-1} + x a_n)\cdots).$$

This will permit evaluating the polnomial using n additions and n multiplications. This method is known as Horner's rule.

(b) In $\log n$ time precompute and store the values

$$x, x^2, x^4 = (x^2)^2, x^8 = (x^4)^2, \ldots, x^{2^j} = \left(x^{2^{j-1}}\right)^2$$

where $j = \lfloor \log n \rfloor$ Note that if $m < n$ then we can write m in binary as $m = \sum_{i=0}^{j} \epsilon_1 2^i$ where $\epsilon_j \in \{0,1\}$ and the ϵ_j can be calculated in $O(j) = O(\log n)$ time. Thus

$$x^m = \prod_{i=0}^{j} \epsilon_i x^{2^i}$$

can be calculated in $O(\log n)$ time using the precomputed squares. This can then be used to evaluate the polynomial in total time $O(k \log n)$ (including the precomputation time).

(DC5) **Interpolating Polynomials** The values $A(x_0), A(x_1), \ldots, A(x_n)$, define a unique degree n polynomial having those values. The Langragian interpolation formula for finding the coefficients $a_0, a_1, \ldots, a_n$ of $A(x)$ works by first setting

$$I_i(x) = \prod_{0 \le j \le n,\, j \ne i} \frac{x - x_j}{x_i - x_j}$$

and then defining

$$A(x) = \sum_i A(x_i) I_i(x).$$

Show how to use the formula to evaluate the coefficients of $A(x)$ in $O(n^2)$ time.

Hints: Note that the $I_i(x)$ are very similar to each other. Instead of constructing them from scratch consider building their smallest common multiple $P(x)$ and then build $I_i(x)$ via division. First recall how long it takes to divide a degree n polynomial by a degree one polynomial. You can use this procedure as a subroutine.

Solution: *Solution: First calculate the values of the coefficients of $P(x) = \prod_{0 \le j \le n} x - x_j$. This can be done by iteratively evaluating*

$$P_j(x) = \prod_{0 \le j \le i} x - x_j = (x - x_j) P_{i-1}(x).$$

Since $P_{j-1}(x)$ is a degree j polynomial and a degree 1 polynomial can be multiplied by a degree i polynomial in $O(j)$ time, finding $P(x) = P_n(x)$ takes $O(\sum_j j) = O(n^2)$ time.

Next note that

$$\prod_{0 \le j \le n,\, j \ne i} x - x_j = \frac{P(x)}{x - x_j}$$

Division of a degree $n+1$ polynomial by a degree one polynomial can be done (basic long division) in $O(n)$ time so, knowing $P(x)$, we can calculate $\prod_{0 \le j \le n,\, j \ne i} x - x_j = \frac{P(x)}{x - x_j}$ in $O(n)$ time for each i. Doing this for all i then needs $O(n^2)$ time.

Also, for fixed i, calculating the scalar value $\prod_{0 \le j \le n,\, j \ne i} x_i - x_j$ takes $O(n)$ time. Do this for all i, using $O(n^2)$ time in total.

Finally, recall that

$$I_i(x) = \prod_{0 \le j \le n,\, j \ne i} \frac{x - x_j}{x_i - x_j} = \frac{\prod_{0 \le j \le n,\, j \ne i} x - x_j}{\prod_{0 \le j \le n,\, j \ne i} x_i - x_j}.$$

Since we have already created the (polynomial) numerator and (scalar) denominator of these terms we can calculate each $I_i(x)$ in $O(n)$ time and all of them in $O(n^2)$ time.

We can then find the coefficients of each $A(x_i)I_i(x)$ in $O(n)$ time or $O(n^2)$ in total. Adding the coefficients together to get the coefficients of the full solution would require a further $O(n^2)$ time

Combining all of the pieces gives an $O(n^2)$ algorithm.

(DC6) Prove from first principles that

$$\begin{array}{rcll} T(n) & = & 1 & \text{if } 1 \le n \le 8 \\ T(n) & = & T\left(\left\lfloor \frac{n}{5} \right\rfloor + 1\right) + T(\left(\left\lceil \frac{3n}{4} \right\rceil + 2\right) + n & \text{if } n > 8 \end{array}$$

satisfies $T(n) = O(n)$.

Note: Before trying to solve this, review the slides describing the analysis of the deterministic selection problem. The analysis of this recurrence should follow the same technique used there.

Solution:

First note that $T(n)$ is well defined, since for every $n > 8$, $T(n)$'s recursive definition only calls $T(n')$ with $n' < n$.

From what we saw in class we know that, because $\frac{1}{5} + \frac{3}{4} < 1$, $T(n) = O(n)$. To prove from first principles, though, we will use the substitution method, guess that $T(n) \le cn$ from some c and work backwards trying to figure out what c is.

Our proof will assume that $T(n)$ is non decreasing in n. This is legal because if not, we can replace $T(n)$ by $\bar{T}(n) = \max_{1 \le i \le n} T(i)$ and show that $\bar{T}(n) = O(n)$.

Let us assume that for base case $n \le b$ we are told that $T(n) \le cn$ (we will choose b later).

The induction step is that we know that for all $n' < n$, $T(n) \le cn$. Then, by induction

$$\begin{aligned} T(n) &\le T\left(\left\lfloor \frac{n}{5} \right\rfloor + 1\right) + T(\left(\left\lceil \frac{3n}{4} \right\rceil + 2\right) + n \\ &\le c\left(\left\lfloor \frac{n}{5} \right\rfloor + 1\right) + c\left(\left\lceil \frac{3n}{4} \right\rceil + 2\right) + n \\ &\le c\left(\frac{n}{5} + 1\right) + c\left(\frac{3n}{4} + 3\right) + n \\ &= \left(\frac{19}{20}c + 1\right) n + 4c \\ &= cn + \left(1 - \frac{c}{20}\right) n + 4c \end{aligned}$$

In order to guarantee that $T(n) \le cn$, it's enough to show that

$$\left(1 - \frac{c}{20}\right) n + 4c \le 0 \tag{3}$$

Note that (1) will be true if and only if

$$n \le c\left(\frac{n}{20} - 4\right) \quad \Leftrightarrow \quad 20n \le c(n - 80) \quad \Leftrightarrow \quad \frac{20n}{n - 80} \le c.$$

Note that if $n > 160$ then setting $40 \le c$ will guarantee $\frac{20n}{n-80} \le c$ and thus (1).

This means that if we are given the base case in the induction that $\forall n \le 160$, $T(n) \le cn$ and we choose $c \ge 40$, then, for all $n > 160$

$$\textit{If } T(n') \le cn' \textit{ for all } n' < n \quad \Rightarrow \quad T(n) \le cn.$$

and thus $T(n) \le cn$ for all n.

In order to satisfy these conditions it suffices to set

$$c = \max\left\{40, \frac{T(1)}{1}, \frac{T(2)}{2}, \ldots, \frac{T(160)}{160}\right\}$$

and we are done.

(DC7) **More Median of Medians** For this problem you can assume the following fact: $\alpha, \beta \ge 0$, N is a non-negative integer and c, D constants (possibly negative). For $n > N$, if

$$T(n) \le T(\alpha n + c) + T(\beta n + d) + \Theta(n)$$

then

$$T(n) = \begin{cases} O(n) & \text{if } \alpha + \beta < 1 \\ O(n \log n) & \text{if } \alpha + \beta = 1 \\ \Omega(n \log n) & \text{if } \alpha + \beta > 1 \end{cases}.$$

Recall that our deterministic selection algorithm yielded the recurrence

$$T(n) = T(n/5) + T(7n/10 + 6) + \gamma n$$

for some constant γ. The formula above implies $T(n) = O(n)$.

Our algorithm (i) splits the items into sets of 5 elements, (ii) found the median of each set and then (iii) found x, the median of those medians. It then ran *partition* with x as a pivot and recursed on the appropriate subset. From the definition of x we were able to prove that the subarrays created by partition both had at most $7n/10 + 6$ elements, leading to the recurrence relation and hence $O(n)$ running time.

Now suppose that instead of splitting the items into sets of size 5, we split them into sets of size 3 and then ran the algorithm the same way. Would we still get an $O(n)$ time algorithm?

What about if we split into sets of size 7? Sets of size 9?

Solution:

If we split into groups of 3 we would have to find the median of $n/3$ items.

The same type of argument that we developed in class would show that after pivoting on that median, recursion throws away at least $\frac{1}{2} \times \frac{2}{3}n - c = \frac{1}{3}n - c$ items (c a small constant) and could retain $\frac{2}{3}n + c$ items.

So the recurrence would be

$$T(n) = T(n/3) + T(2n/3 + c) + O(n)$$

which gives $O(n \log n)$ and not $O(n)$.

If we split into groups of size 7 then we would have to first find the median of $n/7$ items. After using that as the pivot we would throw away $\frac{1}{2} \times \frac{4}{7}n - c' = \frac{2}{7}n - c'$ items; the recursion would retain at most $\frac{5}{7}n + c$ items. So the recurrence would be

$$T(n) = T(n/7) + T(5n/7 + c') + O(n)$$

which only gives $O(n)$.

The general situation is if we replace 5 by an odd $2k+1$ we would throw away $\frac{1}{2} \times \frac{k+1}{2k+1} n - c'$ items, thus retaining at most

$$\left(\frac{2k+1}{2k+1} - \frac{k+1}{2(2k+1)} \right) n + c'' = \left(\frac{3k+1}{2(2k+1)} \right) n + c''.$$

The recurrence relation then becomes

$$T(n) = T\left(\frac{1}{2k+1} n \right) + T\left(\left(\frac{3k+1}{2(2k+1)} \right) n + c'' \right) + O(n)$$

which only gives $O(n)$.

When

$$\frac{1}{2k+1} + \frac{3k+1}{2(2k+1)} < 1$$

this is $O(n)$. But this is true for all $k > 1$, i.e., for splitting into odd groups of any size greater than 3. So the algorithm would work for $5, 7, 9, 11, \ldots$.

Try to figure out what happens when the groups have even size.

(DC8) **Divide and Conquer**

You have found a newspaper from the future that tells you the price of a stock over a period of n days next year. This is presented to you as an array $p[1 \ldots n]$ where $p[i]$ is the price of the stock on day i.

Design an $O(n \log n)$-time divide-and-conquer algorithm that finds a strategy to make as much money as possible, i.e., it finds a pair i, j with $1 \le i < j \le n$, such that $p[j] - p[i]$ is maximized over all possible such pairs.

If there is no way to make money, i.e., $p[j] - p[i] \le 0$ for all pairs i, j with $1 \le i < j \le n$, your algorithm should return "no way".

(a) Describe your algorithm and explain why it gives the correct answer

(b) Analyze your algorithm to show that it runs in $O(n \log n)$ time.

Note: The purpose of this problem is to provide you practice with D & C tools. There is also an $O(n)$ time algorithm for solving this problem. See if you can find it.

Solution: *Note: This problem can be solved using similar ideas as were used for the maximum contiguous subarray problem. Similar to that problem, there is also an $O(n)$ time linear scan solution. That is not what we are asking for here but it is a good exercise to see if you can find it.*

To find the pair i, j in $p[a..b]$ with largest value $p[j] - p[i]$ note that, if q is the midpoint of $a..b$ then exactly one of the following three cases occurs:

- *both i, j are in $[a..q]$*
- *both i, j are in $[q + 1..b]$*
- *i is in $[a..q]$ and j is in $[q + 1..b]$ and*
  $p[i]$ is the smallest value *in $[a..q]$ and $p[j]$ is the largest* value *in $[q + 1..b]$.*

This means that if $b - a > 1$ the optimal solution can be found by finding the maximum of

- *the best solution in $[a, q]$,*
- *the best solution in $[q + 1, b]$*
- *$j_{max} - i_{min}$ where $j_{max} = \max_{q+1 \leq j \leq b} p[j]$ and $i_{min} = \min_{a \leq i \leq q} p[i]$*

If $b - a = 1$ then the ansewr is $\max\{p[b] - p[a], 0$ and if $a = b$ the answer is 0. It is now easy to write a simple divide and conquer algorithm to calculate this.

(b) The procedure calls itself on two subroutines of half the size and does $O(n)$ work for the third case (finding the max and the min of the two sides) and taking the max of the three values. Therefore its running time satisfies

$$T(n) = 2T(n/2) + O(n)$$

which leads to an $O(n \log n)$ solution.

(DC9) **Modified from CLRS**

Professor Olay is consulting for an oil company, which is planning a large pipeline running east to west through an oil field of n wells. From each well, a spur pipeline is to be connected directly to the main pipeline along a shortest path (either north or south). Given the x, y coordinates of the wells, how should the professor pick the optimal location of the main pipeline (the one that minimizes the total length of the spurs). Show that the optimal location can be determined in linear time.

Hint: Try to solve this using the median finding algorithm.

Solution: *Let (x_i, y_i) be the locations of the wells and $f(y)$ be the cost if the pipeline is at latitude y.*

Set $A(y) = \{i,\ y_i > y\}$ and $B(y) = \{i,\ y_i < y\}$ to be the wells above and below y.

Then the cost we need to minimize is the function

$$f(y) = \sum_i |y_i - y| = \sum_{i \in A(y)} (y_i - y) + \sum_{i \in B(y)} (y - y_i).$$

Note that, by definition, $f(y)$ is a piecewise linear continuous function of y.

Now suppose y is changed by value δ. As long as $A(y+\delta) = A(y)$ and $B(y+\delta) = B(y)$

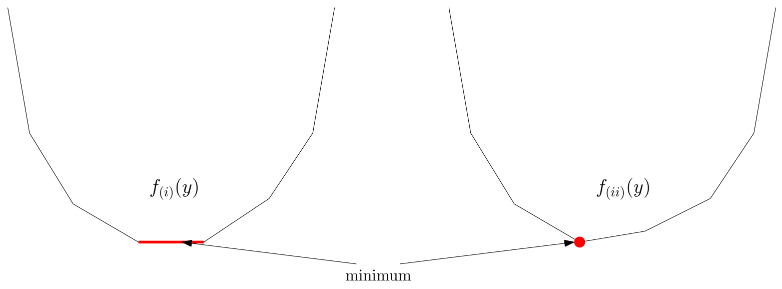
$$f(y+\delta) = f(y) + \delta(|B(y)| - |A(y)|).$$

Since $A(y)$ and $B(y)$ stay constant between two different values of y_i this means that the slope of $f(y)$ in that range is $|B(y)| - |A(y)|$

Note that as y increases $|B(y)|$ is a nondecreasing function and $|A(y)|$ is a non-increasing one and these functions change value only at the y_i. Thus $(|B(y)| - |A(y)|)$ is a nondecreasing function that only changes value at the y_i

You could visualize starting at $y = -\infty$ and increasing y until you find the first y_i such that $A(y_i) > B(y_i)$. The answer would then be in the range $[y_{i-1}, y_i]$ since any y outside that range would have a larger value of $f(y)$. Since the y_i are non-sorted you could use something like the recursive weighted median finding algorithm we developed in the last tutorial (slightly modified) to directly find this y_i. We show another approach below.

From the discussion above, $f(y)$ is a piecewise linear function with increasing slope that starts with slope $-n$ and ends with slope $+n$. Such a function has its minimum (a) everywhere on the line with slope equal 0 if it exists and, if it doesn't (b) at the vertex where the slope flips from negative to positive



Translating this back into our problem, one of the two cases below must occur and whichever does provides the solution.

(a) $\exists y_i, y_j$ *that are successors in sorted order such that* $\forall y,\ y_i < y < y_j,\ B(y) = A(y)$.

(b) $\exists y_i$ *such that* $\forall y > y_i,\ B(y) > A(y)$ *and* $\forall y < y_i,\ B(y) < A(y)$.

Suppose that n is odd. Then the median value of the y_j satisfies (b) and is the solution.

If n is even, let y_i and y_j be the lower and upper medians (the $n/2$'th item and the $(n/2+1)$'st item). If $y_i = y_j$ then case (b) holds again and that value is the solution. If $y_i \neq y_j$ then case (a) holds and the solution is ANY value between y_i and y_j.

Since finding the median can be done on $O(n)$ time, the problem can be solved in $O(n)$ time.

Note too that part of what makes the problem a bit delicate is that there can be many items with the same y value. If all items had different y values then the solution is obviously just the median (or between the two medians).

(DC10) **Modified from CLRS** *Largest i numbers in sorted order.*

Given a set of n numbers, we wish to find the i largest in sorted order using a comparison-based algorithm. Find the algorithm that implements each of the following methods with the best asymptotic worst-case running time and analyze the running times of the algorithms in terms on n and i.

(a) Sort the numbers and list the i largest.

(b) Build a max-priority queue (i.e., a heap) from the numbers and call EXTRACT-MIN i times.

(c) Use a selection algorithm to find the ith largest number, partition around that number and sort the i largest numbers.

Solution:

(a) Sort the numbers and list the i largest.
$O(n \log n + i) = O(n \log n)$

(b) Build a max-priority queue (i.e., a heap) from the numbers and call EXTRACT-MIN i times.
$O(n + i \log n)$

(c) Use a selection algorithm to find the ith largest number, partition around that number and sort the i largest numbers.
$O(n + i \log i)$

Note that the last method is always at least as good as the first two (and sometimes better) and the middle one is always at least as good as the first (and sometimes better).

(DC11) **Finding a "fixed point".**

Input: a sorted array $A[1..n]$ of n **distinct** integers
(Distinct means that there are no repetitions among the integers. The integers can be positive, negative or both).

Design an $O(\log n)$ algorithm to return an **index i such that $A[i] = i$,** if such an i exists. Otherwise, report that no such index exists.

As an example, in the array below, the algorithm would return 4.

i	1	2	3	4	5	6	7	8
A[i]	-3	0	1	4	12	17	20	22

while in the array below this line, the algorithm would return that no such index exists.

i	1	2	3	4	5	6	7	8
A[i]	-3	0	1	7	12	17	20	22

Hint: $O(\log n)$ often denotes some type of binary search. Can you think of any question you might ask that permits you to throw away some constant fraction of the points?

(DC12) **The Majority Problem**

Let $A[1..n]$ be an array of n elements. A *majority element* of A is any element occurring more than $n/2$ times (e.g., if $n = 8$, then a majority element should occur at least 5 times). Your task is to design an algorithm that finds a majority element, or reports that no such element exists.

Suppose that you are not allowed to order the elements; the only way you can access the elements is to check whether two elements are equal or not.

Use standard divide-and-conquer techniques to design an $O(n \log n)$-time algorithm for this problem.

(DC13) **Lagrangian Interpolation Example**

(a) Use Lagrangian Interpolation to construct a degree-2 polynomial $A(x)$ satisfying

$$A(1) = 1, \quad A(2) = 4, \quad A(3) = 9.$$

(b) Use Lagrangian Interpolation to construct a degree-2 polynomial $A(x)$ satisfying

$$A(1) = 1, \quad A(2) = 8, \quad A(3) = 27.$$

(DC*1) **Two for the Price of One Convolutions**

Let $< a_i >$ denote the sequence of real numbers $a_0, a_1, \ldots, a_{n-1}$. The *convolution* of $< a_i >$ and $< b_i >$ both of length n is sequence $< c_i >$ of length $2n-1$ where $c_i = \sum_{j=1}^{i} a_j b^{i-j}$. In class we saw that the FFT could calculate $< c_i >$ in $O(n \log n)$ time.

Now suppose that you are given TWO sequences $< a_i' >$ and $< a_i'' >$ both of length n and $< b_i >$ of length n. Can you see a quick way of calculating the convolution of $< a_i' >$ and $< b_i >$ and the convolution of $< a_i'' >$ and $< b_i >$ without having to run the algorithm twice?

Note: It is not expected that you could solve this yourself without prior knowledge. It is a cute trick that is worth knowing about, though, and understanding it provides a better understanding of what the FFT is really doing.

Solution: *The FFT based algorithm we saw in class did not actually calculate convolutions. It really calculated the coefficients of $C(x) = A(x)B(x)$ where*

$$A(x) = \sum_{j=0}^{n-1} a_j x^j, \quad B(x) = \sum_{j=0}^{n-1} b_j x^j.$$

The important observation is that all the operations were done in the field of complex numbers so the a_j and b_j were allowed to be complex numbers, even if we assumed they were real. Now construct $A'(x)$ and $A''(x)$ as polynomials with the real number coefficients $< a_j >$ and $< a_j'' >$ and set $A(x) = A'(x) + iA''(x)$ where $i = \sqrt{-1}$. Then

$$C(x) = A(x)B(x) = (A'(x) + iA''(x))B(x) = A'(x)B(x) + iA''(x)B(x).$$

where $A'(x)B(x)$ and $A''(x)B(x)$ all have real (non-imaginary) coefficients.

Since the coefficients of $C(x)$ are complex, we can write it as $C(x) = C'(x) + iC''(x)$ where $C'(x)$ and $C''(x)$ all have real coefficients. Then

$$C'(x) = A'(x)B(x) \quad \text{and} \quad C''(x) = A''(x)B(x).$$

so the coefficients of $C'(x)$ are the convolution of $< a_j' >$ and $< b_j >$ and the coefficients of $C''(x)$ are the convolution of $< a_j'' >$ and $< b_j >$.

What this means is that we can actually

- *Run the convolution algorithm on $< a_j' + ia_j'' >$ and $< b_j >$, get the result $< c_j$*
- *The c_j wil be complex numbers so rewrite $c_j = c_j' + ic_j''$.*
- *Return that $< c_j' >$ is the convolution of $< a_j' >$ and $< b_j >$ and $< c_j'' >$ is the convolution of $< a_j'' >$ and $< b_j >$*

and we have solved the problem by just doing one pass and not two.

(DC*2) **An $O(n)$ majority algorithm**

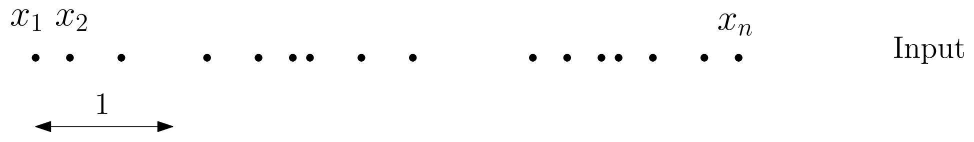
Consider the majority problem defined in problem DC12.

Design an $O(n)$ time algorithm for it.

Greedy (GY)

(GY1) **From CLRS** A Greedy Algorithm.

A *unit-length closed interval* on the real line is an interval $[x, 1+x]$. Describe an $O(n)$ algorithm that, given input set $X = \{x_1, x_2, \ldots, x_n\}$, determines the smallest set of unit-length closed intervals that contains all of the given points. Argue that your algorithm is correct. You should assume that $x_1 < x_2 < \cdots < x_n$.



As an example the points above are given on a line and you are given the length of a 1-unit interval. Show how to place a minimum number of such intervals to cover the points

Solution: *Keep the points in an array. Walk through the array as follows.*

(a) Set $x = x_1$.

(b) Walk through the points in increasing order until finding the first j such that $x_j > x+1$. (if no such point then stop)

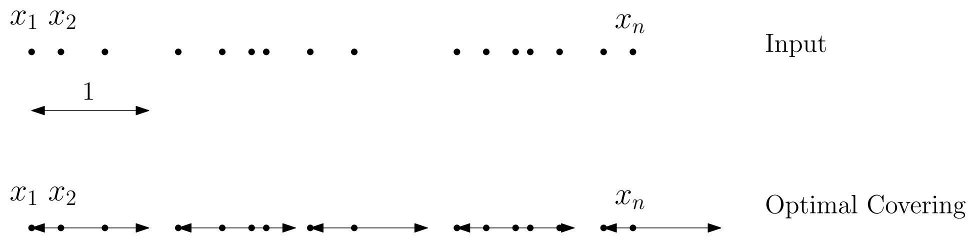
(c) Output $[x, 1+x]$

(d) If there was no such j in (b) then stop. Otherwise, set $x = x_j$.

(e) Go to Step (b)

Note that each point is seen only once so this is an $O(n)$ algorithm.

This is the solution for the points in the question:



We must now prove correctness.

Let Greedy(i,k) be the algorithm run on the Array $[i..j]$. Note that it can be rewritten as

(a) Output $[x_i, 1+x_i]$

(b) Find min j such that $x_j > x_i + 1$.

(c) If such a j does not exist, stop, else return Greedy(j,k).

We will prove correctness by induction on the number of points $|X|$.

If $|X| = 1$ the algorithm is obviously correct. Otherwise, suppose $|X| = n$ and we know that the algorithm is correct for all problems with size $< n$.

Let $O[i,j]$ to be the minimum number of intervals needed to cover $\{x_i, \ldots, x_j\}$ and $G[i,j]$ the number of intervals Greedy uses to cover them.

We assume that $[x_1, 1+x_1]$ does not cover all of X because, if it did, Greedy would return that one interval solution which is optimal.

Let j be the smallest index such that $x_j > 1 + x_1$. Note that Greedy returns $[x, 1 + x]$ concatenated with the greedy solution for $\{x_j, \ldots, x_n\}$ and, by induction, its Greedy solution for $\{x_j, \ldots, x_n\}$ is optimal. So, it uses $1 + O(j, n)$ intervals.

Now, suppose that there is a solution OPT different than the Greedy one. Let $[x, 1+x]$ be the interval with the leftmost starting point in OPT. Note that $x \le x_1$ because otherwise x_1 would not be covered by any interval in OPT. Let k be the minimum index such that $x_k > 1+x$. After removing $[x, 1+x]$ the remaining intervals in OPT must form an optimal solution for $\{x_k, \ldots, x_n\}$ (otherwise we could build a solution using fewer intervals). So the total number of intervals used by OPT is $1 + O(k, n)$.

The main observation is that because $x \le x_1$, $j \ge k$. Thus the optimal solution for $\{x_k, \ldots, x_n\}$ is A solution for $\{x_j, \ldots, x_n\}$ so it has at least as many intervals as the optimal solution for $\{x_j, \ldots, x_n\}$, i.e., $O(k, n) \ge O(j, n)$.

Combining the pieces yields

$$\begin{aligned} G(1,n) &= 1 + G(j,n) \\ &= 1 + O(j,n) \\ &\le 1 + O(k,n) \\ &= O(1,n) \end{aligned}$$

which means that Greedy must be optimal for X.

(GY2) Consider the problem of making change for n cents using the fewest number of coins. Assume that each coin's value is an integer.

(a) Describe a greedy algorithm to make change consisting of quarters (25 cents), dimes (10), nickels (5), and pennies (1). Prove that your algorithm yields an optimal solution.

(b) Suppose that the available coins are in denominations that are powers of c. i.e. the denominations are $c^0, c^1, ..., c^k$ for some integers $c > 1$ and $k \ge 1$. Show that the greedy algorithm always yields an optimal solution.

(c) Give a set of coin denominations for which the greedy algorithm does not yield an optimal solution. Your set should include a penny so that there is a solution for every value of n.

Solution:

(a) Set $c_q = \lfloor n/25 \rfloor$,
This is the largest number of quarters that can be used to make change for n cents.
Set $n_q = n - 25c_q$.
This is the amount remaining after using c_q quarters
Set $c_d = \lfloor n_q/10 \rfloor$ (largest number of dimes that can be used)
Set $n_d = n_q - 10c_d$ (amount remaining)

Set $c_n = \lfloor n_d/5 \rfloor$
Set $c_p = n_p = n_d - 5c_n$.

Greedy solution uses c_q *quarters,* c_d *dimes,* c_n *nickles and* c_p *pennies.*

Since $n_q < 25$, $\mathbf{c_d \le 2}$. *Similarly,* $n_2 < 10$, *so* $\mathbf{c_n \le 1}$.
Furthermore, $10c_d + 5c_n \le n_q < 25$ *so* $(\mathbf{c_d}, \mathbf{c_n}) \ne (\mathbf{2}, \mathbf{1})$.
Finally, $\mathbf{c_p} = (\mathbf{n} \bmod \mathbf{5}) \le \mathbf{4}$.

Let O *be an optimal solution using* o_q *quarters,* o_d *dimes,* o_n *nickels and* o_p *pennies. We will show that* $(o_q, o_d, o_n, o_p) = (c_q, c_d, c_n, c_p)$.
First note that if $o_p \ge 5$ *every 5 pennies can be replaced with one nickle, reducing the number of coins used, so we can assume* $o_p \le 4$. *Furthermore, all the other coins are mutiples of 5. Thus* $o_p = (n \bmod 5) = c_p$.

If $o_d \ge 3$ *replace every three dimes with* 1 *quarter and* 1 *nickle without increasing the number of coins. So, we can assume* $o_d \le 2$.
If $o_n \ge 2$ *replace every 2 nickels with one dime, reducing the number of coins used, so we can assume* $o_n \le 1$.

If $o_d = 2$ *and* $o_n = 1$ *then these three coins can be replaced by one quarter, reducing the number of coins used, contradictng optimality. Thus* $10o_d + 5o_n \le 20$ *which means that* $10o_d + 5o_n + o_p \le 25$ *so* $o_q \ge \lfloor n/25 \rfloor$. *But the max number of quarters that can be used is* $c_q = \lfloor n/25 \rfloor$. *So* $o_q = c_q$.

We have now seen that $(o_q, o_p) = (c_q, c_p)$. *This means that*

$$10o_d + 5o_n = n - 25o_q - o_p = n - 25c_q - c_p = 10c_d + 5c_n$$

amd by construction $n - 25c_q - c_p < 25$.
It remains to show that $(o_d, o_n) = (c_d, c_n)$.
From the previous discussion:
$c_d \le 2,\ c_n \le 1$ *and* $(c_d, n_n) \ne (2,1)$;
$o_d \le 2,\ o_n \le 1$ *and* $(o_d, o_n) \ne (2,1)$.
There are only five possible cases for both (c_d, c_n) *and* (o_d, o_n) : $(0,0)$, $(0,1)$, $(1,0)$, $(1,1)$, $(2,0)$. *Since each of these is a unique amount of money (corresponding uniquely to* $0, 5, 10, 15, 20$ *cents)* $(o_d, o_n) = (c_d, c_n)$.

(b) Recall that there is a unique way to write n *in base* c*, i.e.,*

$$n = \sum_i a_i c^i \quad \textit{such that} \quad \forall i,\ 0 \le a_i < c.$$

First consider the greedy solutions. Suppose it chooses g_i *coins of type* c^i. *If, for some* i, $g_i \ge c$ *then the greedy solution could have chosen one more coin of denomination* c^{i+1}. *So, for all* i, $g_i < c$, *and by the uniqueness of such a representation, for all* i, $g_i = a_i$.
Now let o_i *denote the number of* c^i *coins used in an optimal solution for making change of* n *cents. Note that,* $\forall i$, $o_i < c$; *if this was not true then we could replace*

the o_i coins of size c_i with one coin of size c^{i+1} and $(o_i - c)$ coins of size c_i, reducing the number of coins used, contradicting optimality of O.

Thus, for all i, $o_i < c$ which means that, for all i, $o_i = a_i$.

Since, for all i, $g_i = a_i = o_i$, greedy IS optimal.

(c) *Let $1, 4, 6$ be the set of coin denominations.*

Suppose we make change for $n = 8$ cents.

The greedy solution uses one 6 cent coin and two 1 cent coins, i.e. it uses 3 coins.

However, the optimal solution would only use two 4 cents coins.

(GY3) (CLRS–16.2-4) Professor Midas drives an automobile from Newark to Reno along Interstate 80. His car's gas tank, when full, holds enough gas to travel m miles, and his map gives the distance between gas stations on his route. The professor wishes to make as few gas stops as possible along the way. Give an efficient method by which Professor Midas can determine at which gas stations he should stop and prove that your algorithm yields an optimal solution.

Solution: *The simple greedy algorithm is optimal. It is to drive to the furthest possible city that one can reach with the current gas in the car and then fill up the tank and then continue.*

Suppose that the input to the problem is the $n + 1$ city locations $0 = x_0 < x_1 < \ldots < x_n$ with 0 being the starting location (Reno), $x_n = m$ being the destination location (Newark) and all of the other x_i being locations of gas stations.

Let GREEDY be the greedy solution which we will denote by G. We will prove optimality of greedy by induction on n. Let O be any optimal solution and assume that Greedy is optimal on all problems on size $< n$ (for the basis this is obviously true on sets of size 1 and 2). We may also assume that, whenever O adds gas, O fills the gas tank completely (since this can not make the solution worse). We use $|O|$ and $|G|$ to denote the numbers of stops each solution makes.

Now consider the input on n points. Let g_1 be the first stop that Greedy makes and o_1 be the first stop that OPT makes. By the definition of Greedy, $o_1 \le g_1$. Write

$$\begin{aligned} G &= g_1, g_2, \ldots, g_k \\ O &= o_1, o_2, \ldots, o_{k'} \end{aligned}$$

By definition, $k' \le k$, where the g_i and o_i are the stops the algorithms make.

Now let $t = \max\{i : o_i \le g_1\}$. From the observations above we know that $t \ge 1$.

Set

$$O' = g_1, o_{t+1}, o_{t+2}, \ldots, o_{k'}.$$

Since $g_1 \ge o_t$, this is a legal tour. Thus $t = 1$, otherwise O was not optimal. So

$$O' = g_1, o_2, o_3, \ldots, o_{k'}.$$

Since $|O'| = k' = |O|$, O' is also optimal. Now note that $o_2, o_3, \ldots, o_{k'}$ must be an optimal stopping pattern for the problem $x_{g_1}, x_{g_1+1}, \ldots, x_n$ because otherwise we could replace it in O with a smaller set of gas fills, getting a smaller optimal solution (which is imposisble).

From the induction hypothesis we know that $g_2, \ldots, g_k$ *is an optimal stopping pattern for the problem* $x_{g_1}, x_{g_1+1}, \ldots, x_n$.

Thus $k = k'$ *and greedy is optimal for our original set.*

(GY4) Huffman Coding

From the book Problems on Algorithms, *by Ian Parberry, Prentice-Hall, 1995.*

Build a Huffman Tree on the frequencies $\{1, 3, 5, 7, 9, 11, 13\}$.

For this problem, follow the rule that if two items are combined in a merge, the smaller one goes to the left subtree (in case of ties *within* a merge you can arbitrarily decide whic goes on the left).

Are there any *ties* in the Huffman Construction process, i.e., are there times when the merge procedure can choose between different choices of items?

How many *different* Huffman Trees can be built on this frequency set?

Solution: *There is one possible tie. There is a time when the smallest items are* $7, 9$ *and there are two possible* $9s$ *to choose from.*

That is the only *time when the algorithm can make an arbitary choice, so only two different Huffman codes can be built off of this frequency set.*

(GY5) The goal of the interval partitioning problem (i.e., the classroom assignment problem) taught in lecture was to open as few classrooms as possible to accommodate all of the classes. The greedy algorithm taught, sorted the classes by starting time.

It then ran through the classes one at a time; at each step it first checked if there was an available empty classroom. Only if there was no such classroom would it open a new classroom.

Prove that if the classes are sorted by finishing time the algorithm might not give a correct answer.

Note: Proving that something does not work usually means to find a counter-example.

(GY*1) A Huffman Coding Variant

(a) Recall that in each step of Huffman's algorithm, we merge two trees with the lowest frequencies. Show that the frequencies of the two trees merged in the ith step are at least as large as the frequencies of the trees merged in any previous step.

(b) Suppose that you are given the n input characters, already sorted according to their frequencies. Show how you can now construct the Huffman code in $O(n)$ time. (*Hint:* You need to make clever use of the property given in part (a). Instead of using a priority queue, you will find it advantageous to use a simpler data structure.)

Solution:

(a) We will show that the frequencies merged at the $(i+1)$st step are at least as large as the frequencies merged at the i'th step. The proof will folllow.

Let $a \leq b \leq c \leq d$ be the values of the four smallest frequencies in the priority queue immediately before the start of the ith step. The ith step merges a, b and creates a new frequency $z = a + b$ that is inserted into the priority queue.

In the $(i+1)$st step, only two possibilities can occur

(a) $z \geq d$: In this case the two frequencies merged are values c, d and the statement is trivially correct.

(b) $z < d$: In this case, the two values merged are c, z and the statement is still correct.

(b) From part (a) we know that the new frequencies created by the Huffman algorithm are created in non-decreasing order.

The algorithm is as follows:

- *Create two* queues *(not priority queues). Recall that a queue is a linked list that permits checking and/or removing its "head" item in $O(1)$ time and adding an item to its "tail" in $O(1)$ time.*
- *Add the original items in sorted order to the 1st queue in $O(n)$ time. Note that removing items from the head of this queue, one at a time, will remove them in sorted order.*
  In the algorithm, we will only remove items from this queue but never add new ones.
- *Every time a new frequency is created, add it to the tail of the 2nd queue in $O(1)$ time. Note that, from (a), we know that the items will be added in nonincreasing order to the queue (so they will be sorted in THAT queue).*

- *At every step of the Huffman algorithm the current set of frequencies is stored in the 2 queues. The items are* sorted *in each queue. Find the item with smallest frequency by comparing the items at the heads of the two queues. Note that this can be done in $O(1)$ time because the head of each queue always hold the smallest item in that queue. Remove that smallest item from its corresponding queue. Now repeat the $O(1)$ operation to find the smallest remaining frequency in the two queues and remove it.*
- *add the newly created frequency to the tail of the 2nd queue.*

This takes the two smallest items off at every step and adds the new item so it is exactly implementing the Huffman algorithm. It uses $O(1)$ time per step instead of $O(n)$ so it is an $O(n)$ algorithm in total.

To understand the above it is useful to look at an example. Suppose that the input frequencies are $Q_0 = \{2, 2, 3, 4, 5, 6, 7, 8, 12, 13\}$. The weights that Huffman coding would maintain in sorted order are

$$\begin{aligned}
Q_1 &= \{3, \mathbf{4}, 4, 5, 6, 7, 8, 12, 13\} \\
Q_2 &= \{\mathbf{4}, 5, 6, 7, \mathbf{7}, 8, 12, 13\} \\
Q_3 &= \{6, 7, \mathbf{7}, 8, \mathbf{9}, 12, 13\} \\
Q_4 &= \{\mathbf{7}, 8, \mathbf{9}, 12, 13, \mathbf{13}\} \\
Q_5 &= \{\mathbf{9}, 12, 13, \mathbf{13}, \mathbf{15}\} \\
Q_6 &= \{13, \mathbf{13}, \mathbf{15}, \mathbf{21}\} \\
Q_7 &= \{\mathbf{15}, \mathbf{21}, \mathbf{26}\} \\
Q_8 &= \{\mathbf{26}, \mathbf{36}\} \\
Q_9 &= \{\mathbf{52}\}
\end{aligned}$$

where the weights in bold face are the ones created by merging.

Instead of maintaining just one queue the algorithm maintains two Queues. Q_i will be the remaining original items in their original sorted order. Q'_i will be the created weights, with the newest created weight inserted at the back of Q'_i. Note that, at at any given time Q'_i is sorted (from part (a)). The algorithm can always find the two smallest items by choosing the two smallest from the two smallest in Q_i and the two smallest in Q'_i.

$$\begin{aligned}
Q_0 &= \{2, 2, 3, 4, 5, 6, 7, 8, 12, 13\} & Q'_0 &= \emptyset \\
Q_1 &= \{3, 4, 5, 6, 7, 8, 12, 13\} & Q'_1 &= \{\mathbf{4}\} \\
Q_2 &= \{5, 6, 7, 8, 12, 13\} & Q'_2 &= \{\mathbf{4}, \mathbf{7}\} \\
Q_3 &= \{6, 7, 8, 12, 13\} & Q'_3 &= \{\mathbf{7}, \mathbf{9}\} \\
Q_4 &= \{8, 12, 13\} & Q'_4 &= \{\mathbf{7}, \mathbf{9}, \mathbf{13}\} \\
Q_5 &= \{12, 13\} & Q'_5 &= \{\mathbf{9}, \mathbf{13}, \mathbf{15}\} \\
Q_6 &= \{13\} & Q'_6 &= \{\mathbf{13}, \mathbf{15}, \mathbf{21}\} \\
Q_7 &= \emptyset & Q'_7 &= \{\mathbf{15}, \mathbf{21}, \mathbf{26}\} \\
Q_8 &= \emptyset & Q'_8 &= \{\mathbf{26}, \mathbf{36}\} \\
Q_9 &= \emptyset & Q'_9 &= \{\mathbf{51}\}
\end{aligned}$$

(GY*2) Extra problem. Huffman Coding and Megersort.

Recall that Mergesort can be represented as a tree with each internal node corresponding to a merge of two lists.

The weight of a leaf is 1;
the weight of an internal node is he sum of the weights of its two children,
or equivalently, the number of leaves in its subtrees.

The cost of a single Merge is the number of items being merged, so the cost of Mergesort is the sum of the weights of the tree's internal nodes.

(a) Prove that the cost of Mergesort can be rewritten as the weighted external path length of its associated tree, when all leaves have weight 1

(b) Prove that the recursive Mergesort studied in class has height $h = \lceil \log_2 n \rceil$, with $x = 2^h - n$ leaves on level $h - 1$ and $n - x$ leaves on level h.

(c) Show that an optimal Huffman tree for n items, all with the same frequency 1, will have the property that the tree will have height $h = \lceil \log_2 n \rceil$, with $x = 2^h - n$ leaves on level $h - 1$ and $n - x$ leaves on level h.

(d) Use the above facts to prove that recursive mergesort is *optimal*, i.e., that there is no other merge pattern for merging n items that has lower total cost.

Solution:

(a) *This is not specific to mergesort. We wil prove that it is always true that if we define the weight of an internal node to be the number of leaves in its subtrees then the sum of the weights of the internal nodes is the weighted external path length of its associated tree, when all leaves have weight 1.*

In what follows, let T be a tree, $h(t)$ its height, $L(t)$ the number of leaves in the tree, $W(T)$ the sum of the weights of the internal nodes in the tree and $EPL(T)$ the weighted external path length of the tree when all leaves have weight 1. We want to prove that $W(T) = EPL(T)$ for all trees T.

This statement is proven by induction on the height h of the tree. It is obviously true when $h = 1$ (and the tree has one or two leaves).

Suppose that it is true for all trees of height $< h$. Let T be a tree with $h(T) = h$ and x be the root of T.

If x has only one child let T_1 be the subtree falling off of x. Then by definition $W(T) = L(T)+W(T_1)$. On the other hand, every node in T is exactly one level deeper than it was in T_1 so $EPL(T) = EPL(T_1)+L(T_1)$. Since $h(T_1) = h(T)-1 = h-1 < h$ the proof follows by the induction hypothesis and noting

$$W(T) = W(T_1) + L(T) = EPL(T_1) + L(T_1) = EPL(T).$$

If x has two children then let T_1 and T_2 be the left and right tree falling off of x. Since $h(T_1), h(T_2) < h(T) = h$ the induction hypothesis tells us that $W(T_1) = EPL(T_1)$ and $W(T_2) = EPL(T_2)$.

By definition $W(T) = W(T_1) + W(T_2) + L(T)$. On the other hand, every node in T_1 and T_2 is exactly one level deeper in T than it was in T_1 or T_2. So

$$EPL(T) = EPL(T_1) + L(T_1) + EPL(T_2) + L(T_2) = EPL(T_1) + EPL(T_2) + L(T)$$

and the proof again follows from the induction hypothesis and noting

$$W(T) = W(T_1) + W(T_2) + L(T) = EPL(T_1) + EPL(T_2) + L(T) = EPL(T).$$

(b) *The proof will be by induction on i. We will prove, for every i the statement is true for all $n \le 2^i$.*

It is obviously true for all $n \le 2^1 = 2$. Now suppose that it is true for all $n \le 2^{i-1}$.

Let $2^i < n \le 2^{i+1}$. We need to show that it is true for all such n.

There are two cases, n odd and n even.

(a) n even: Then $n = 2n'$ with $n' \le 2^i$. The algorithm splits the n items into two sets, each of size n', building a tree on each of them. By the induction hypothesis, each of those trees has height $h' = \lceil \log_2 n' \rceil$, with $x' = 2^{h'} - n'$ leaves on level $h' - 1$ and $n' - x'$ leaves on level h'.

Note that, by definition the height of the final tree is

$$h' + 1 = \lceil \log_2 n' \rceil + 1 = \lceil \log_2 2n' \rceil = \lceil \log_2 n \rceil$$

proving the first part of the statement.

Note that the leaves of the final tree are exactly the leaves of the subtree but pushed one level deeper. this means that the subtree has

$$2x' = 2(2^{h'} - n) = 2^{h'+1} - 2n' = 2^h - n$$

leaves on level $h' - 1 + 1 = h - 1$ and $n - x$ leaves on level $h' + 1 = h$.

(b) n odd: The proof is similar but needs a little bit of extra work when $n = 2^{i-1} + 1$.

(c) *First note that the Huffman tree T must (by definition) have minimal weighted external path length among all trees with n nodes (with all leaves having weight 1).*

Note that this immediately implies that all leaves must be on the bottom two levels of the Huffman tree. Suppose not and the tree has height h. Let (i) x be a leaf on level $d \leq h-2$, (ii) u be a leaf on level h and (iii) p be u's parent on level $h-1$.

Let v be the sibling of v (which must exist because the tree is full). Now make p be a leaf (associated with the character that used to be associated with x. This removes u, v.

Further, make x be an internal node with two children, associated with the characters that used to be associated with u, v. We have now created a new prefix-free coding tree for the same set of characters which has SMALLER weighted external path length, contradicting the optimality of T.

We now know that T is full (i.e., every internal node has two children; this fact was proven in class) and has all of its children at depths h and $h-1$. There are two cases
(a) ALL of the leaves are on one level, i.e., h. This means that all of the 2^{h-1} nodes on level $h-1$ are internal nodes. Since each of them must have two children and both of those children must be leaves there are $2 \cdot 2^h$ leaves (all on level h in the tree) so $n = 2^h$ and the statement is trivially correct.
(b) Leaves exist on BOTH level h and $h-1$. Note that, by the fullness, T has 2^{h-2} internal nodes at depth $h-2$ and thus 2^{h-1} nodes on level $h-1$. Suppose it has $y > 0$ leaves on level $h-1$. Note that because there are also leaves on level h, we must have $0 < y < 2^{h-1}$ and all of the non-leaf nodes on level $h-1$ have two children on level h. This implies that there are $2^{h-1} - y$ internal nodes on level $h-1$ and $2(2^{h-1} - y) = 2^h - 2y$ leaves on level h. This means $n = 2^h - 2y + y = 2^h - y$ so $2^{h-1} < n < 2^h$ and $h = \lceil \log_2 n \rceil$. Furthermore, the number of leaves on level $h-1$ is $y = 2^h - n$ and the number in level h is $n - y$ proving the 2nd part of the statement.

(d) *(d) follows directly from (a), (b) and (c).*

The Huffman algorithm constructs a tree with minimum weighted external path length. If all characters have weight 1, (c) immediately implies that this tree has cost

$$(h-1)(2^h-n)+h(2n-2^h)$$

where $h = \lceil \log_2 n \rceil$. By the proof of optimality of Huffman codes, we know that this is the mimimum weighted external path length such a tree could have.

From (a) we know that the cost of a Mergesort is equal to the weighted external path length of its tree. From the previous paragraph we know that this can not be less than

$$(h-1)(2^h-n)+h(2n-2^h)$$

From (b) we know that the tree for recursive Mergesort has exact cost

$$(h-1)(2^h-n)+h(2n-2^h)$$

which therefore implies it must be optimal.

Graphs (GR)

(GR1) Let $G = (V, E)$ be an undirected graph where V is the set of vertices and E is the set of edges.
Assume that there are no self-loops or duplicated edges.
Answer all questions below as a function of $|V|$, the number of vertices.

a) What is the maximum number of edges in G?

b) What is the maximum number of edges in G if two vertices have degree 0.

c) What is the maximum number of edges that an acyclic graph G can have?

d) What is the minimum number of edges in G if G is a connected graph and contains at least one cycle?

e) What is the minimum possible degree a vertex in a connected graph G can have?

f) What is the maximum length of any simple path in G?

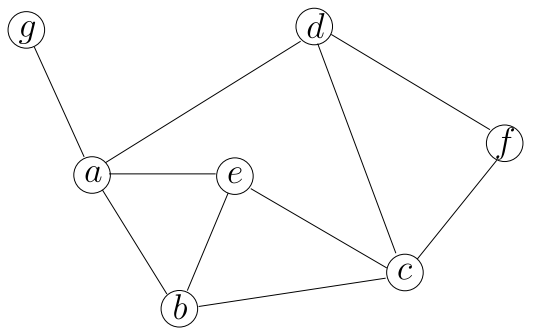
(GR2) Let $G = (V, E)$ be a connected undirected graph. Prove that

$$\log(E) = \Theta(\log V).$$

Note: we implictly use this fact in many of our analyses in class.

(GR3) The adjacency list representation of a graph G, which has 7 vertices and 10 edges, is:

$$\begin{array}{ll} a :\rightarrow d, e, b, g & b :\rightarrow e, c, a \\ c :\rightarrow f, e, b, d & d :\rightarrow c, a, f \\ e :\rightarrow a, c, b & f :\rightarrow d, c \\ g :\rightarrow a & \end{array}$$



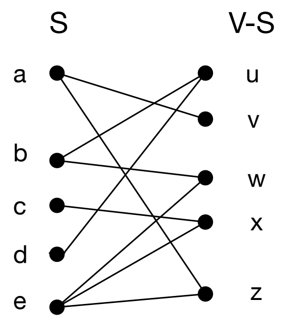
(a) Show the breadth-first search tree that is built by running BFS on graph G with the given adjacency list, using vertex a as the source.

(b) Indicate the edges in G that are NOT in the BFS tree in part (a) by dashed lines.

(c) Show the depth-first search tree that is built by running DFS on graph G with the given adjacency list, using vertex a as the source.

(d) Indicate the edges in G which are NOT in the DFS (c) by dashed lines.

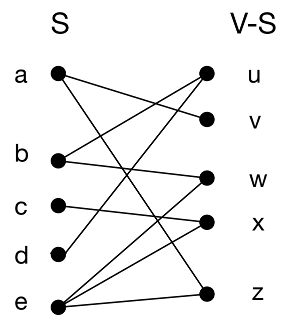
(GR4) An (undirected) graph $G = (V, E)$ is *bipartite* if there exists some $S \subset V$ such that, for every edge $\{u, v\} \in E$, either
(i) $u \in S$, $v \in V - S$ or
(ii) $v \in S$, $u \in V - S$.

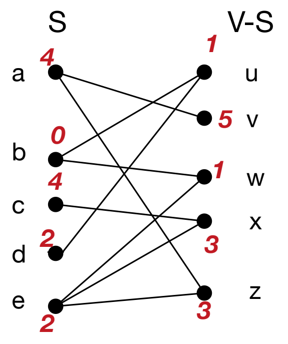


Let $G = (V, E)$ be a connected graph. Design an $O(|V| + |E|)$ algorithm that checks whether G is bipartite. *Hint: Run BFS.*

Solution: *Run BFS from any vertex. Recall that $d[v]$ will store the shortest distance from the root to v. Set S to be the set of all vertices with $d[v]$ even.*

The main observation is that
(i) G will be bipartite if and only if (ii) all edges (u, v) in the graph satisfy that the parity of $d[v]$ and $d[u]$ are not the same, i.e., $d[v]$ is odd and $d[u]$ is even or vice versa.



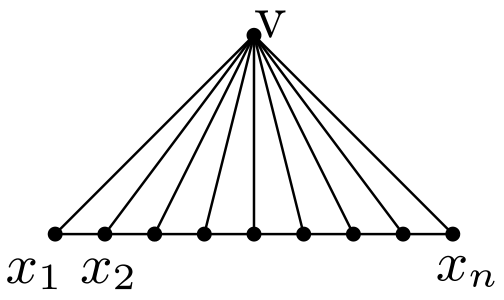


A) First, assume (ii). Then it is easy to see that G is, by definition bipartite. Just set S to be the set of all even vertices. So $(i) \Rightarrow (ii)$.

B) Now, assume (ii). Let $S, V - S$ be a bipartite split and assume without loss of generality that some specific $b \in S$. By the definition of bipartite, the length of every *path from b to other nodes in S is even and the length of every path to nodes in $V - S$ is odd. In particular the lengths of the* shortest *paths to nodes in S are even and to those in $V - S$ are odd. So, the partity of the endpoints of all edges in G must be different. So $(ii) \Rightarrow (i)$.*

(GR5) In the Fan Graph F_n, node v is connected to all the nodes and the other connections are given by the adjacency lists below.

$$v : x_1, x_2, \ldots, x_n, \qquad x_1 : v, x_2$$
$$x_n : v, x_{n-1} \qquad \forall i \neq 1, n, \quad x_i : v, x_{i-1}, x_{i+1}$$

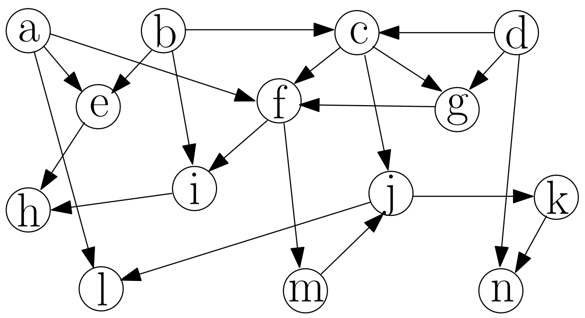


(a) : Describe the tree that is output when BFS is run on F_n starting from initial vertex v; (ii) initial vertex x_1;
(iii) x_n; (iv) Other x_i.

(b) : Describe the tree that is output when DFS is run on F_n starting from initial vertex v; (ii) initial vertex x_1;
(iii) x_n; (iv) Other x_i.

"Describing the tree" means sketching the structure of the tree and also writing down a general formula for $v.p$ (the parent of node v in the BFS/DFS tree) for all v.

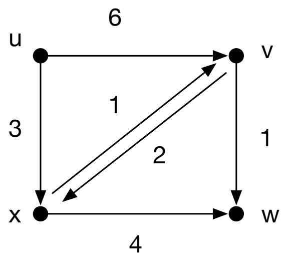
(GR6) Give a topological ordering of the following graph.



Solution: *There are many possible topological orderings. Here's one:*

$$a, b, d, e, c, g, f, m, I, j, h, l, k, n.$$

(GR7) Execute Dijkstra's algorithm on the following digraph, where u is the source vertex.



You need to indicate only the following:

(a) the order in which the vertices are removed from the priority queue.

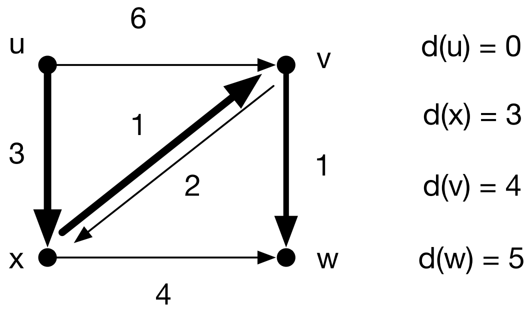
(b) the final distance values $d[]$ for each vertex.

(c) the different distance values $d[]$ assigned to vertex b, as the algorithm executes.

Solution:

Dijkstra's algorithm with u as the source vertex.

x is taken off first, then v and then w.



(GR8) Suppose that instead of using a heap to store the tentative vertex distances, Dijkstra's algorithm just kept an array in which it stored each vertex's tentative distance.

It then finds the next vertex by running through the entire array and choosing the vertex with lowest tentative distance.

What would the algorithm's running time be?

Is this better than our implementation for some graphs?

Solution:

Let n be number of vertices and m number of edges.

The algorithm uses $O(n^2)$ time (independent of m):

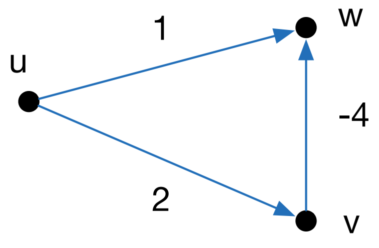
At each step, it uses $O(n)$ time to scan through all of the remaining vertices.
It also uses $O(m)$ time to update the tentative distances (since the cost of one update is $O(1)$ in changing an array value, rather than $O(\log n)$ for a decrease key).

This is "better" than the $O((n+m)\log n)$ implementation we learned in class if the graph has $m = \Omega\left(\frac{n^2}{\log n}\right)$. For example, if the graph has $m = \Theta(n^2)$ then Dijkstra's original version runs in $O(n^2)$ time while the version taught in class uses $O(n^2 \log n)$ time.

Note: This is actually the "original" implementation of Dijkstra's algorithm by Dijkstra. The use of priority queues to save time came later.

(GR9) Give a simple example of a directed graph with negative-weight edges for which Dijkstra's algorithm produces incorrect answers. Why does the correctness proof of Dijkstra's algorithm not go through when negative-weight edges are allowed?

Solution: *Consider the graph below with the algorithm starting at u. w is the next vertex chosen and is given distance 1 from u. But, the real shortest distance from u to w is -2, by taking the path uvw.*



The correctness proof falls apart at various junctures. One of the fundamental ones is that the proof assumed that a full path can not have shorter cost than any of its subpaths but this is no longer true when negative edges are allowed.

(GR10) Let $G = (V, E)$ be a connected undirected graph in which all edges have weight either 1 or 2. Give an $O(|V| + |E|)$ algorithm to compute a minimum spanning tree of G. Justify the running time of your algorithm. (*Note:* You may either present a new algorithm or just show how to modify an algorithm taught in class.)

Solution:

You could run a Prim's algorithm, just implementing your Priority Queue differently. Every priority queue operation except for the initialization (which will take $O(|V|)$ time) will now take $O(1)$ time so the algorithm will run in $O(|E| + |V|)$ time.

To do this, note that since every value in the priority queue is either ∞, 1 or 2 you can implement the priority queue by maintaining 3 doubly linked lists. List 0 contains all items with key value ∞, List 1, those with key values 1 and List 2 those with key values 2.

When you start put everyone is in List 0 because they all have key value ∞. You can create this priority-queue in $O(|V|)$ time.

To implement Extract-Min: If List 1 contains a value, just pull off the first one. Otherwise, if List 2 is not empty, pull the first value off of List 2. Otherwise, pull the first item off List 0. Extract Min takes $O(1)$ time.

To implement Decrease Key: (a) take the item out of its current list which can be done in $O(1)$ time (because the list is doubly linked). (b) Insert the item into the front of the appropriate new list. This can be done in $O(1)$ time because there are only two such possible lists.

(GR11) Let G be a connected undirected graph with weights on the edges. Assume that all the edge weights are distinct. Prove from first principles that G only has one (unique) MST.

Solution: *Let $T = \{e_1, e_2, \ldots, e_{n-1}\}$ be some MST.*

Now remove $e_i = (u_i, v_i)$ from T. This leaves two connected subsets $(S, V - S)$ with $u_i \in S$ and $v_i \in V - S$.

i) **Note that the uniqueness of the edge weights ensure that there is a unique lightest edge crossing $(S, V - S)$. We claim that e_i is this unique lightest edge.**

Suppose not and there was some edge e' crossing $(S, V - S)$ with $w(e') < w(e_i)$. Then adding e' to $T - \{e_i\}$ connects S and $V - S$. $T' = T \cup \{e'\} - \{e_i\}$ is a tree with cost

$$w(T') = w(T) - w(e_i) + w(e') < w(T)$$

contradicting that T was a MST.

ii) **Let T' be ANY MST. We claim that $e_i \in T'$.**

Suppose not. Then since T' is connected it contains some path P_i from u_i to v_i. Since $u_i \in S$ and $v_i \in S - V$, P_i must cross *the $(S, V - S)$ cut at least once, i.e., it contains some edge $e' = (u'_i, v'_i)$ with $u'_i \in S$ and $v'_i \in V - S$. From part (i) we have that $w(e_i) < w(e')$.*

Note that $P_i \cup \{e_i\}$ forms a cycle. Thus, adding e_i to T' creates a cycle with P_i. Removing ANY edge from P_i will leave a connected graph with $n - 1$ edges, i.e, a tree. In particular,

$$T'' = T' \cup \{e_i\} - \{e'\}$$

will be a tree with cost

$$w(T'') = w(T') - w(e') + w(e_i) < w(T')$$

contradicting the fact that T' is supposed to be a MST.

Combining parts (i) and (ii) we find that EVERY edge $e_i \in T$ is in EVERY MST. But, since an MST has exactly $n-1$ edges, every MST is exactly T.

(GR12) Let G be a connected undirected graph with distinct weights on the edges, and let e be an edge of G.

Suppose e is the largest-weight edge in some cycle of G.

Show that e cannot be in the MST of G.

Solution: *Let T be the MST of G and suppose T contains $e = (u, v)$. Removing e from T breaks it into two connected components S and $V - S$ with $u \in S$ and $v \in V - S$.*

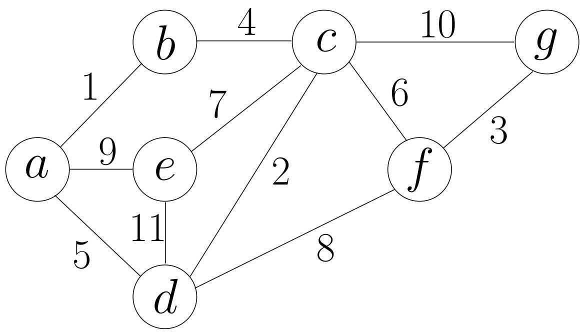
Consider the cycle that has e as the largest-weight edge. This cycle starts from u, goes to v, and takes another path to go back to u. This path must cross this cut somewhere via an edge $e' = (u', v')$ with $u' \in S$ and $v' \in V - s$. Since e was the largest weight edge on the cycle, $w(e') < w(e)$.

Now we add e' to T. This connects the two components so it is now a spanning tree T'. T' is T with e removed and e' added so

$$cost(T') = cost(T) - w(e) + w(e') < cost(T)$$

contradicting fact that T was a MST.
$\Rightarrow$ e can not be the largest edge on some cycle.

(GR13) Prim's minimum spanning tree algorithm and Dijkstra's shortest path algorithm are very similar, but with crucial differences. Run both algorithms on the following graph, and show the partial MST / shortest path tree after every new edge has been added. The starting vertex for both algorithms is "a".



(GR14) Let $G = (V, E)$ be a weighted graph with non-negative distinct edge weights.

Now replace every weight $w(u, v)$ with its square $(w(u, v))^2$.

(a) Is T still a MST of G with the new weights? Either prove that it is or give a counterexample

(b) Next consider a shortest path $u \to v$ in the original graph. Is this path still a shortest path with the new weights? Either prove that it is or give a counterexample

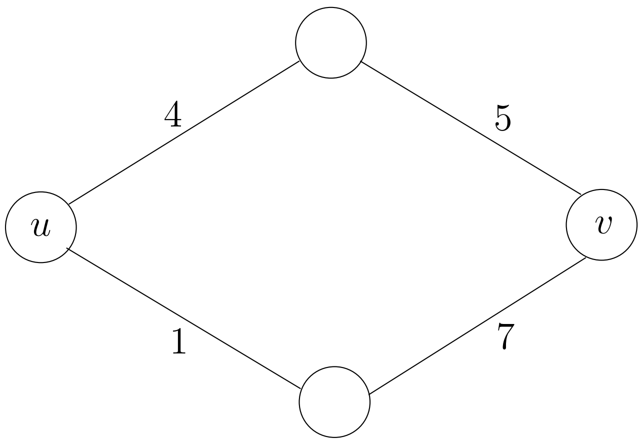
Solution:

(a) Yes, the MST remains unchanged when the weights are changing from w to w^2. There are many proofs of this statement. Here's one

Run Kruskal's algorithm on the edges. The first step of Kruskal's algorithm is to sort the edges by weight. Squaring the weights of the edges does not change their order so Kruskal's algorithms on the new weights will start with the same ordering as with the old weights.

After the original sorting step, Kruskal's algorithm never looks at the weights again; it just looks at which edges connect to which vertices. So, the second part of the algorithm will run the same for both the original and squared edge weights which means that it will output the same MST.

(b) No, it may not be a shortest path in the new graph. In the graph below the shortest path from u to v goes through the top but after squaring the weights it goes through the bottom.



(GR15) Path Reliability

Suppose that we are given a cable network of n sites connected by duplex communication channels. Unfortunately, the communication channels are not perfect.
The channel between sites u and v is known to fail with (given) probability $f(u, v)$

The probabilities of failure for different channels are known to be mutually independent events.

One of the n sites is the central station and your problem is to compute the most reliable paths from the central station to all other sites (i.e., the paths of lowest failure probabilities from the central station to all other sites).

Design an algorithm for solving this problem, justify its correctness, and analyze its time and space complexities.

Solution:

Let $f'(u,v) = 1 - f(u,v)$ which is the probability of edge (u,v) NOT failing.

The probabilty of a path $P = u_0, u_1, u_2, \ldots, u_t$ NOT failing is then the product of the edge non-failures, i.e.,

$$Cost(P) = \prod_{i=0}^{t-1} f'(u_i, u_{i+1}).$$

Let u_0 be the central station. Our goal is to find, for every v, a path from u_0 to v with MAXIMAL cost.

Finding maximum $Cost(P)$ is the same as finding minimum $\frac{1}{Cost(P)}$ which is the same as finding the path with minimum $\log \frac{1}{Cost(P)}$.

Now set $w(u,v) = -\log f'(u,v) = \log \frac{1}{f'(u,v)}$ and recall that the log of the product of values is the sum of the logarithms of the values so

$$\log \frac{1}{Cost(P)} = \sum_{i=0}^{t-1} \log \frac{1}{f'(u_i, u_{i+1})} = \sum_{i=0}^{t-1} w(u_i, u_{i+1}).$$

Putting all the pieces together we have just shown that finding a max cost path from u to v *is exactly the same as finding a shortest path* from u to v with edge distances $w(u,v)$.

So, we can use Dijkstra's single source shortest path algorithm to solve the problem in $O(|E| \log |V|)$ time.

Final Note: One issue that we did not explicitly discuss is that Dijkstra's algorithm only works for edges that have non-negative weights.

This is not a problem, though, because the $f'(u,v)$ are all probabilities between 0 *and* 1 *Thus, the $w(u,v)$ are all non-negative and we can apply Dijkstra's algorithm.*

This transformation via log*s of a* max-weight path *where weight is* product of edge weights *to a* shortest path*, where path length is sum of edge lengths, is very well know and commonly used.*

(GR16) Let $T = (V, E)$ be a tree and $e = (u,v) \in E$.

Show that removing e from T leaves a graph with exactly two connected components with one component containing u and the other containing v.

(GR17) It is not difficult to see that if e is a minimum weight edge in G then e is always an edge in *some* Minimum Spanning Tree for G.

Prove that if e is a maximum weight edge in G, the corresponding statement is not correct.

This means proving that it is possible that e *does* belong to a MST of G, i.e., provide a counterexample. (Note: It is also possible that e does not belong to any MST for G.)

Dynamic Programming

(DP1) Give an $O(nk)$-time dynamic programming algorithm that makes change using the minimal possible number of coins.

The solution obviously depends upon the country you are in.

Let the local given coin denominations be $d_1 < d_2 < \cdots d_k$, where $d_1 = 1$ (which guarantees that some solution always exists).

Solution:

This problem has the optimal substructure property. If we knew that an optimal solution for j dollars used **a** *coin of denomination d_i, it would have to be true that, in that solution, the change for the remaining $j - d_i$ dollars would have to be an optimal (minimum) set of coins for that subproblem, i.e., $c[j] = 1 + c[j - d_i]$ As base cases, we have that $c[j] = 0$ for all $j \leq 0$.*
To develop a recursive formulation, we have to check all denominations, giving

$$c[j] \quad = \quad \begin{cases} 0 & \text{if } j \leq 0, \\ 1 + \min_{1 \leq i \leq k}\{c[j - d_i]\} & \text{if } j > 1. \end{cases}$$

We can compute the $c[j]$ values in order of increasing j by using a table. The following procedure does so, producing a table $c[1..n]$

Note: The code avoids explicitly examining $c[j]$ for $j \leq 0$ by checking $j \geq d_i$ before looking up $c[j - d_i]$.

The procedure also produces a table $denom[1..n]$, where $denom[j]$ is the denomination of the "first" coin used in an optimal solution to the problem of making change for j dollars.

COMPUTE-CHANGE(n, d, k)

```
for $j \leftarrow 1$ to $n$
    $c[j] \leftarrow \infty$
    for $i \leftarrow 1$ to $k$
        if $j \geq d_i$ and $1 + c[j - d_i] < c[j]$
            $c[j] \leftarrow 1 + c[j - d_i]$
            $denom[j] \leftarrow d_i$
return c and denom
```

This procedure obviously runs in $O(nk)$ time

We use the following procedure to output the coins used in the optimal solution computed by COMPUTE-CHANGE:

GIVE-CHANGE$(j, denom)$

```
if $j > 0$
    give one coin of denomination $denom[j]$
    GIVE-CHANGE($j - denom[j], denom$)
```

The initial call is GIVE-CHANGE$(n, denom)$. Since the value of the first parameter decreases in each recursive call, this procedure runs in $O(n)$ time.

(DP2) KFCC is considering opening a series of restaurants along the highway. The n available locations are along a straight line; the distances of these locations from the start of the Highway are given in miles and in increasing order: $m_1, m_2, \ldots, m_n$. The constraints are as follows:

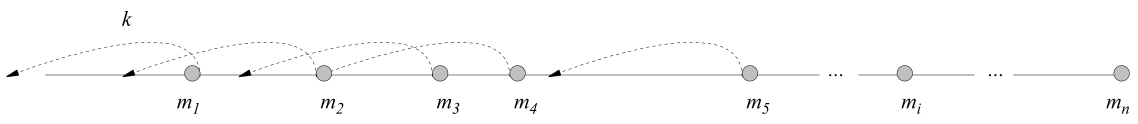
(a) At each location, KFCC may open at most one restaurant.
The expected profit from opening a restaurant at location i is p_i, where $p_i > 0$ and $i = 1, 2, \ldots, n$.

(b) Any two restaurants should be at least k miles apart,
where k is a given positive integer.

Give a dynamic programming algorithm that determines a set of locations at which to open restaurants that maximizes the total expected profit earned.

medskip

Solution:



We define $T[i]$ to be the total profit from the best valid configuration using only locations from within $1, 2, \ldots, i$.

We also store $R[i]$ which is 1 if there is a restaurant at location i and 0 otherwise.

Case 1: Base case *If $i = 0$, then there is no location available to choose from to open a restaurant. So $T[0] = 0$.*

Case 2: General case *If $i > 0$, then we have two options.*

(a) Do not open a restaurant at location i
If no restaurant opened at location i, then the optimal value will be optimal profit from valid configuration using only location $1, 2, \ldots, i-1$. This is just $T[i-1]$.

(b) Open a restaurant at location i. Opening a restaurant at location i, gives expected profit p_i. After building at location i, the nearest location to the left where a restaurant can be built is c_i, where

$$c_i = \max\{j \leq i \,:\, m_j \leq m_i - k\}.$$

To obtain a maximum profit, we need to obtain the maximum profits from the remaining locations $1, 2, \ldots, c_i$. This is just $p_i + T[c_i]$.

Since these are the only two possibilities, we derive the following rule for constructing table T:

$$T[i] = \begin{cases} 0, & \text{if } i = 0 \\ \max\{T[i-1], p_i + T[c_i]\}, & \text{if } i > 0 \end{cases}$$

If $T[i] = T[i-1]$, then $R[i] = 0$; and $R[i] = 1$ otherwise.

Note: This immediately gives an algorithm that $O(n)$ PLUS the time required to calculate the c_j.

This gives an $O(n^2)$ algorithm if we use brute force to find every c_i in $O(n)$ time.

A little more thought shows that we could find $c_i = \max\{j : m_j \le m_i - k\}$ in $O(\log n)$ time using binary search. This would give us a $O(n \log n)$ algorithm.

The extra twist in this solution is that we will see soon that it's possible to calculate all of the c_i in $O(n)$ time, which will allow a total $O(n)$ time algorithm.

Note that for some values of i and c_i may not exist in which case, we set $c_i = 0$.

Algorithm to find optimal profit and locations to open restaurants.
Assumes c_i are known.

```
Find-Optimal-Profit-And-Pos($m_1, \ldots, m_n, p_1, \ldots, p_n, c_1, \ldots, c_n,$)
1: $T[0] = 0$
2: for $i = 1$ to $n$ do
3:     Not-Open-At-i = $T[i-1]$;  Open-At-i = $p_i + T[c_i]$
4:     if Not-Open-At-i > Open-At-i then
5:         $T[i]$ =Not-Open-At-i;  $R[i]$ =0
6:     else
7:         $T[i]$ =Open-At-i; $R[i]$ =1
8:     end if
9: end for
10: return $T[n]$ and R;
```

Algorithm to compute $c_i = \max\{j : m_j \le m_i - k\}$ for every i

Uses fact that $0 = c_1 \le c_2 \le \cdots \le c_n < n$.
So, after finding c_{i-1}, starts searching for c_i at $j = c_{i-1}$;
  increments j until finds correct value of c_i.

Note that algorithm runs in $O(n)$ time because line 5 can only be implemented $O(n)$ times!
Can visualize this as having "j" being a pointer testing for current c_i;
pointer can only move right ...

Compute-ci$(m_1, \ldots, m_n, k)$

```
1: $c_1 = 0$;
2: for $i = 2$ to $n$ do
3:    $j = c_{i-1}$
4:    while ($m_{j+1} \le m_i - k$) do
5:       $j = j + 1$
6:    end while
7:    $c_i = j$
8: end for
9: return $c_1, \ldots, c_n$
```

Algorithm to report optimal locations to open restaurants

Report-Optimal-Locations$(R, c_1, \ldots, c_n,)$

```
1: $j = n$; $S = \emptyset$
2: while $j \ge 1$ do
3:    if $R[j] = 1$ then
4:       Insert $m_j$ into $S$; $j = c_j$
5:    else
6:       $j = j - 1$
7:    end if
8: end while
9: return $S$;
```

- **Compute-ci** takes $O(n)$ time to compute c_i for every i.
- Find **Optimal-Profit-And-Pos** takes $O(n)$ time to compute T and R.
- **Report-Optimal-Locations** takes $O(n)$ time to report the optimal locations for opening restaurants along the Highway.

Therefore, the overall running time is $O(n)$.

(DP3) Give an $O(n^2)$ time dynamic programming algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers, i.e, each successive number in the subsequence is greater than its predecessor.

For example, if the input sequence is $\langle 5, 24, 8, 17, 12, 45 \rangle$, the output should be either $\langle 5, 8, 12, 45 \rangle$ or $\langle 5, 8, 17, 45 \rangle$.

Hint: Let $d[i]$ be the length of the longest increasing subsequnce whose last item is item i.

Solution: *We first give an algorithm which finds the length of the longest increasing subsequence; later, we will modify it to report a subsequence with this length.*

Let $X_i =< x_1, \dots, x_i >$ denote the prefix of X consisting of the first i items. Define $c[i]$ to be the length of the longest increasing subsequence that ends with x_i. It is clear that the length of the longest increasing subsequence in X is given by $\max_{1 \le i \le n} c[i]$.

The longest increasing subsequence that ends with x_i has the form $< Z, x_i >$ where Z is the longest increasing subsequence that ends with x_r for some $r < i$ and $x_r \le x_i$. Thus, we have the following recurrence relation:

$$c[i] = \begin{cases} 1 & \textit{if } i = 1 \\ 1 & \textit{if } x_r > x_i \textit{ for } 1 \le r < i \\ \max_{\substack{1 \le r < i \\ x_r < x_i}} c[r] + 1 & \textit{if } i > 1 \end{cases}$$

(a) The basis follows from the fact the longest increasing subsequence in a sequence consisting of one number is the number itself.

(b) The recurrence relation says that if all the numbers to the left of i are greater than x_i then the length of the longest increasing subsequence ending in x_i is 1.

(c) Otherwise, the length of the longest increasing subsequence ending in x_i is 1 more than the length of the longest increasing subsequence ending at a number x_r to the left of x_i such that x_r is less than x_i.

We store the $c[i]$'s in an array whose entries are computed in order of increasing i. After computing the c array we run through all the entries to find the maximum value. This is the length of the longest increasing subsequence in X.

In order to **report** *the optimal subsequence we need to store for each i, not only $c[i]$ but also the value of r which achieves the maximum in the recurrence relation. Denote this by $r[i]$.*

We can trace the solution as follows.
Let $c[k] = \max_{1 \le i \le n} c[i]$.
Then x_k is the last number in the optimal subsequence.
The second to last number is $x_{r[k]}$, the third to last number is $x_{r[r[k]]}$ and so on until we have found the all the numbers of the optimal subsequence.

Running Time: *Since it takes $O(i)$ time to compute the i-th entry of the c array, the total time to compute the c array is $O(\sum i) = O(n^2)$. It takes $O(n)$ time to find the maximum in the c array. Finally, the time to trace the solution is $O(n)$. Thus, the running time is dominated by the time it takes to compute the c array, which is $O(n^2)$.*

(DP4) The subset sum problem is: Given a set of n positive integers, $S = \{x_1, x_2, \ldots, x_n\}$ and an integer W determine whether there is a subset $S' \subseteq S$, such that the sum of the elements in S' is equal to W.

For example, if $S = \{4, 2, 8, 9\}$ and $W = 11$, then the answer is "yes" because there is a subset $S' = \{2, 9\}$ whose elements sum to 11. If $W = 7$. the answer is "no".

Give a dynamic programming solution to the subset sum problem that runs in $O(nW)$ time. Justify the correctness and running time of your algorithm.

Solution:

The solution is to construct a boolean array $A[i,j]$, $0 \le i \le n$ and $0 \le j \le W$, defined as follows:

$$A[i,j] = \begin{cases} \textit{true} & \textit{If there is a subset of } \{x_1, x_2, \ldots, x_i\} \textit{ that sums to } j \\ \textit{false} & \textit{Otherwise} \end{cases}$$

To construct a recurrence we observe the following:

Basis: $A[i,0] = \textit{true}$, $0 \le i \le n$,
because given 0 or more items, you can always form the sum 0 by picking no item. Also, $A[0,j] = \textit{false}$, $1 \le j \le W$, because if there are no items to pick from, then we cannot form any sum > 0.

Last weight too large: $A[i,j] = A[i-1,j]$ *if* $i > 0$ *and* $x_i > j$.
The solution cannot contain x_i if x_i exceeds j, the sum to be formed. Therefore the sum j can be formed using a subset of $\{x_1, x_2, \ldots, x_i\}$ if and only if it can be formed using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$.

Last weight not too large:
$A[i,j] = (A[i-1, j-x_i]$ *OR* $A[i-1,j])$, *if* $i > 0$ *and* $j \ge x_i$.
This follows from the following observations. If sum j can be formed using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$, then either this subset includes item x_i or it does not. If it includes item x_i then it should be possible to form the sum $j - x_i$ using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$; otherwise if it does not include item x_i then it should be possible to form the sum j using a subset of $\{x_1, x_2, \ldots, x_{i-1}\}$.

Combining these observations yields the following recurrence relation:

$$A[i,j] = \begin{cases} \textit{true} & \textit{if } 0 \le i \le n \textit{ and } j = 0 \\ \textit{false} & \textit{if } i = 0 \textit{ and } 1 \le j \le W \\ A[i-1,j] & \textit{if } i > 0 \textit{ and } x_i > j \\ A[i-1,j-x_i] \textit{ OR } A[i-1,j]) & \textit{if } i > 0 \textit{ and } j \ge x_i \end{cases}$$

The algorithm takes as inputs the sum to be formed W, the number of items n, and the sequence $x = x_1, x_2, \ldots, x_n$.

It stores the $A[i,j]$ values in a table $A[0 \ldots n, 0 \ldots W]$ whose values are computed in order of increasing i (note that for any given i it does not matter in which order we compute the $A[i,j]$'s).

Following this order ensures that the table entries used to compute $A[i,j]$ have all been computed before *the algorithm evaluates $A[i,j]$. At the end of the computation, $A[n,W]$ is true, if there is a subset that sums to W, otherwise it is false.*

```
Dynamic-SubsetSum($x, n, W$)
    $A[0,0] = true$
    for $j = 1$ to $W$ do
        $A[0,j] = false$
    for $i = 1$ to $n$ do
        $A[i,0] = true$
        for $j = 1$ to $W$ do
            if $x_i > j$ then
                $A[i,j] = A[i-1,j]$
            else $A[i,j] = A[i-1, j-x_i]$ OR $A[i-1,j]$
```

Running Time: *Since the table has $O(nW)$ entries and it takes constant time to compute any one entry, the total time to build the table is $O(nW)$. The total running time is $O(nW)$.*

(DP5) The (Restricted) Max-Sum Problem.

Let A be a sequence of n numbers $a_1, a_2, \ldots, a_n$.

Find a subset S of A that has the maximum sum, provided that, if $a_i \in S$, then $a_{i-1} \notin S$ and $a_{i+1} \notin S$.

Note that, unlike in the previous question, A is a sequence in which order matters (and not an unordered set).

As an example, if $A = 1, 8, 6, 3, 7$, the max possible sum is $S = \{8, 7\}$.

(DP6) Give an $O(nW)$ dynamic programming algorithm for the 0-1 knapsack problem where n is the number of items and W is the max weight that can fit into the knapsack. Recall that the input is i items with given weights $w_1, w_2, \ldots, w_n$ and associated values $v_1, v_2, \ldots, v_n$ and the objective is to choose a set of items with weight $\leq W$ with maximum value.

Now suppose that you are given *two* knapsacks with the same max weight. Give an $O(nW^2)$ dynamic programming algorithm for finding the maximum value of items that can be carried by the two knapsacks.

Note: The one-knapsack problem is taught in the COMP3711 lecture.

Solution: *The implicit assumption in this problem is that W and the w_i are all integers. This was not needed for the fractional knapsack case (and its greedy solution) but is required for the* 0-1 *knapsack problems.*

We first solve the one knapsack case.

The algorithm is based on defining a table

$$V(i,w), \quad 0 \leq i \leq n, \quad 0 \leq w \leq W$$

in which $V(i,w)$ is the maximum value of objects from the set of the first i objects *that can be placed in a knapsack that has maximum weight w. The optimal solution to the problem is $V(n,W)$.*

The algorithm is based on the following recurrence relation:

$$V(i,w) = \max\Big(V(i-1,w),\, V(i-1,w-w_i)+v_i\Big)$$

The initial conditions are $\forall i$, $V(i,w) = -\infty$ if $w < 0$ and $\forall w \ge 0$, $V(0,w) = 0$.
Note that a value of $-\infty$ is essentially being used as a flag for something being impossible.

The basic idea behind the equation is that there are two possible cases for the optimal knapsack of size w using the first i items. Either the i'th item is not included or the i'th item is included.

If the i'th item is not included,
then the optimal solution is the optimal solution using the first $i-1$ items, which has value $\mathbf{V(i-1,w)}$.

If the i'th item is included,
then it adds value v_i. After including it, the knapsack still has weight capacity of $w-w_i$ and this needs to optimally filled by the first $i-1$ items. The best way of doing this has value $V(i-1,w-w_i)$. Adding the two pieces together gives $\mathbf{V(i-1,w-w_i)+v_i}$.

Note that if $w_i > w$ the i'th item can't fit into the knapsack so this option is not possible. This is flagged in the recurrence by the fact that $V(i-1,w-w_i) = -\infty$. An alternative option is to write the recurrence as

$$V(i,w) = \begin{cases} \max\Big(V(i-1,w),\, V(i-1,w-w_i)+v_i\Big) & \text{if } w_i \le w \\ V(i-1,w) & \text{if } w_i > w \end{cases}$$

Given the recurrence we can fill in the recurrence table by, for each fixed $i = 1,2,\ldots,n$ (in increasing order), calculating $V(i,w)$ from the recurrence for every $w = 1,2,3\ldots,W$. In this order, when it's time for $V(i,w)$ to be calculated, both of $V(i-1,w)$ and $V(i-1,w-w_i)$ are already known.

There are $O(nW)$ table entries and each requires only $O(1)$ time to evaluate so the entire algorithm uses only $O(nW)$ time.

The above calculates the best Value. *To find the set of items that achieves that value you will need to keep an auxiliary matrix $Included(i,w)$ which is set to be* false *or* true, *depending upon whether the max occurs at $V(i-1,w)$ or $V(i-1,w-w_i)+v_i$. Using our standard approach we can reconstruct the optimal set from this matrix by working backwards from $Included(n,W)$.*

We now discuss the case of two knapsacks.

The algorithm for this case is a simple generalization of the previous one and is based on defining a table

$$V(i,w^1,w^2), \quad 0 \le i \le n,\ 0 \le w^1 \le W,\ 0 \le w^2 \le W$$

in which $V(i,w^1,w^2)$ is the maximum value of objects from the set of the first i objects *that can be placed in two knapsacks, the first one having weight capacity w^1, and the second having weight capacity w^2. The optimal solution to the problem is $V(n,W,W)$.*

The algorithm is based on the following recurrence relation:

$$V(i,w^1,w^2) = \max\big(V(i-1,\, w^1,\, w^2),\ V(i-1,\, w^1-w_i,\, w^2)+v_i,\ V(i-1,\, w^1,\, w^2-w_i)+v_i,\big)$$

(with initial conditions $\forall i,\ V(i,\, w^1,\, w^2) = -\infty$ *if* $w^1 < 0$ *or* $w^2 < 0$ *and* $\forall w^1,\, w^2 \geq 0$, $V(0,w^1,w^2) = 0$.

The basic idea behind the equation is is that the three terms on the right hand side correspond to the three cases in which the optimal solution for $V(i,w^1,w^2)$
(i) does not use item i *at all,*
(ii) puts item i *in the first knapsack and*
(iii) puts item i *in the second knapsack.*

We do not go into more details because this is very similar to the derivation in the previous case.

Notice that, if all of the items on the right hand side were already known, then the left hand side could be calculated in $O(1)$ *time. It's not hard to find an ordering that satisfies this (which?) so the algorithm runs in* $O(nW^2)$.

As before, this algorithm only finds the best *value. To find the actual items in the two knapsacks you will need to keep an auxiliary matrix that associates with each entry in the* $V(...)$ *matrix how the optimal value for that entry was achieved.*

(DP7) (Problem from an old exam)

Let $''A'' \to 1$, $``B'' \to 2$, $\ldots$, $``Z'' \to 26$.

Given an encoded message M containing n digits in $1\ldots 9$, design an $O(1)$-time dynamic programming algorithm to determine the total number of ways to decode M.

As an example, 15243 can be decoded in 4 different ways, "AEBDC", "AEXC", "OBDC", and "OXC".

(DP8) A sequence of numbers $a_1, a_2, \ldots, a_n$ is *oscillating* if

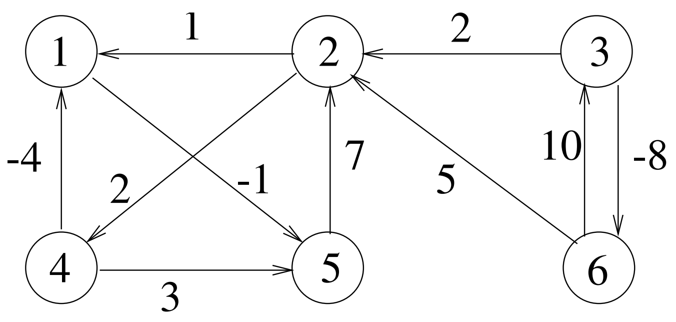
$$a_i < a_{i+1} \text{ for every odd index } i \quad \text{and} \quad a_i > a_{i+1} \text{ for every even index } i.$$

For example, the sequence $2, 7, 1, 8, 2, 6, 1, 8, 3$ is oscillating.

Describe and analyze an efficient dynamic programming algorithm to find a longest oscillating subsequence in a sequence of n integers.

(DP9) Use the DP approach to design an $O(n)$ time algorithm for solving the maximum contiguous subarray problem.

(DP10) Run the Floyd-Warshall algorithm on the weighted, directed graph shown in the figure. Show the matrix $D^{(k)}$ that results for each iteration of the outer loop.



Solution:

$$D^{(0)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & \infty & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & 3 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix} \quad D^{(1)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ \infty & 2 & 0 & \infty & \infty & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ \infty & 7 & \infty & \infty & 0 & \infty \\ \infty & 5 & 10 & \infty & \infty & 0 \end{bmatrix}$$

$$D^{(3)} = D^{(2)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ 1 & 0 & \infty & 2 & 0 & \infty \\ 3 & 2 & 0 & 4 & 2 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 8 & 7 & \infty & 9 & 0 & \infty \\ 6 & 5 & 10 & 7 & 5 & 0 \end{bmatrix}$$

$$D^{(4)} = \begin{bmatrix} 0 & \infty & \infty & \infty & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & \infty & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D^{(5)} = \begin{bmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ 0 & 2 & 0 & 4 & -1 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

$$D^{(6)} = \begin{bmatrix} 0 & 6 & \infty & 8 & -1 & \infty \\ -2 & 0 & \infty & 2 & -3 & \infty \\ -5 & -3 & 0 & -1 & -6 & -8 \\ -4 & 2 & \infty & 0 & -5 & \infty \\ 5 & 7 & \infty & 9 & 0 & \infty \\ 3 & 5 & 10 & 7 & 2 & 0 \end{bmatrix}$$

(DP11) (CLRS) Give an algorithm that takes as input a directed graph with positive edge weights, and returns the cost of the shortest cycle in the graph (if the graph is acyclic, it should say so). Your algorithm should take time at most $O(n^3)$, where n is the number of vertices in the graph.

Solution:

In $O(n^3)$ time run the Floyd-Warshall algorithm to find all the values $d_{i,j}$, the costs of the min cost path from v_i to v_j. The answer to the problem is

$$A = \min_{i,j}(d_{i,j} + d_{j,i})$$

Let OPT be the real cost of a minimum cost cycle. We need to show that $A = OPT$.

First note that every term $d_{i,j} + d_{j,i}$ is the cost of some cycle (start at i, follow the path of length $d_{i,j}$ to j and then follow the path of length $d_{j,i}$ back to i.). Since A is the minimum of some set of cycle costs, $OPT \leq A$.

Now suppose that we know some min-cost cycle C. *Let i', j' be any two points on the cycle. Let $d'_{i',j'}$ and $d'_{j',i'}$ be the costs on that cycle for the paths from i' to j' and from j' to i'. By definition*

$$d'_{i',j'} \geq d_{i',j'} \quad \text{and} \quad d'_{i',j'} \geq d_{i',j'}$$

Thus

$$OPT = cost(\mathrm{C}) = d'_{i',j'} + d'_{j',i'} \geq d_{i,j} + d_{j,i} \geq A$$

Thus, $A = OPT$.

The running time is $O(n^3)$ for the Floyd-Warshall algorithm and $O(n^2)$ for the rest, so the full running time is $O(n^3)$.

(DP12) Assume that all edges have positive weight. Design an algorithm that will, for every pair of vertices, count the *number* of shortest paths between that pair.

Solution: *If zero weight cycles existed then the number of shortest paths could be infinite. The reason for requiring all edges to have positive weight was to guarantee that no zero-weight cycles exist. This will guarantee that all shortest paths are simple and that there will thus be a finite number of shortest paths.*

The short version of the solution is just to modify Floyd-Warshall slightly. Recall that Floyd-Warshall maintains a variable $d^{(k)}_{i,j}$ which is the length of the shortest path from i to j such that all intermediate vertices on the path (if any) are in the set $\{1, 2, ..., k\}$. Add another variable $N^{(k)}_{i,j}$ which will be the number of shortest paths from i to j such that all intermediate vertices on the path (if any) are in the set $\{1, 2, ..., k\}$.

Recall that the update step in the code, when moving from phase $k-1$ to phase k was to set $d^{(k)}_{i,j} = min\{d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}\}$. We still do this but we now also update $N^{(k)}_{i,j}$ as follows based on which of the three cases below occurs:

Case 1:

$$d^{(k-1)}_{i,j} > d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j}.$$

If this is the case then all *shortest paths with intermediate vertices in $\{1, 2, ..., k\}$ must have k as an intermediate vertex (exactly once) so set*

$$N^{(k)}_{i,j} = N^{(k-1)}_{i,k} \cdot N^{(k-1)}_{k,j}.$$

Case 2:

$$d_{i,j}^{(k-1)} < d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}.$$

Since k now doesn't appear as an intermediate vertex,

$$N_{i,j}^{(k)} = N_{i,j}^{(k-1)}.$$

Case 3:

$$d_{i,j}^{(k-1)} = d_{i,k}^{(k-1)} + d_{k,j}^{(k-1)}.$$

In this case, the shortest path from i to j can be achieved by both using and not using k as an intermediate vertex so the answer is

$$N_{i,j}^{(k)} = N_{i,j}^{(k-1)} + N_{i,k}^{(k-1)} \cdot N_{k,j}^{(k-1)}.$$

This gives a $O(n^3)$ algorithm. The space can be reduced from $O(n^3)$ down to $O(n^2)$ in the same way as in the regular Floyd-Warshall algorithm.

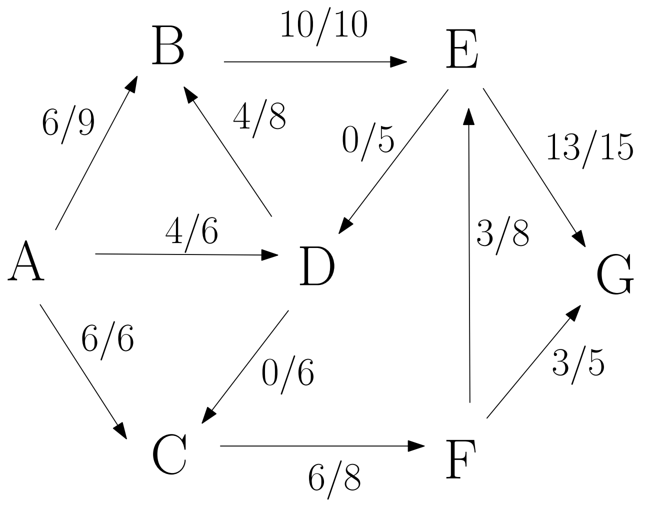
(DP13) In the class notes on the Floyd-Warshall Algorithm we said that it was possible to reduce the space requirement from $O(n^3)$ to $O(n^2)$ by not keeping each of the n $n \times n$ matrices $D^{(i)}$ but instead keeping only ONE matrix and reusing it.

We then wrote the code for doing that.

Why does this space-reduced code work and give the correct answer?

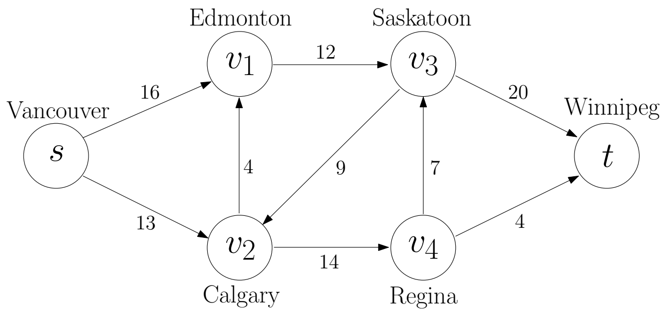
Max Flow and Matchings

(MM1) Consider the given graph with flow values f and capacities c (f/c) as shown. $s = A$ and $t = G$.



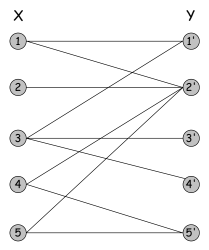
Draw, the residual graph.
Find an augmenting path.
Show the new flow created by adding the augmenting path flow.
Is your new flow optimal?
Prove or disprove.

(MM2) Show the execution of the Edmonds-Karp algorithm on the following flow network.



Recall that the Edmonds-Karp algorithm is to implement Ford-Fulkerson by always choosing a *shortest path* (by number of edges) in the current residual graph.

(MM3) Find a Maximum Bipartite Matching in the graph below using the Max-Flow Method taught in class.



(MM4) Find Find Stable Matchings based on these preference lists.

Man	1st	2nd	3rd	4th
A	c	a	b	d
B	a	d	c	b
C	d	a	b	c
D	d	b	a	c

Woman	1st	2nd	3rd	4th
a	B	D	C	A
b	A	C	D	B
c	B	D	C	A
d	A	D	C	B

(1) What is the Man-Optimal matching?

(2) What is the Woman-Optimal matching?

(3) Are they the same?

(MM5) **Multisource and Multisink**

Note: This problem is only intended for COMP3711. COMP3711H saw this already in class. Also be aware that the mathematical formulations used for the max flow problem are slightly different for COMP3711 and COMP3711H. This was written using the COMP3711 formulations.

Max-Flow as taught in class assumed a single source and single sink.

Suppose the flow network has multiple sources $s_1, s_2, \ldots, s_m$ and multiple sinks $t_1, t_2, \ldots, t_n$ and the goal is to move as much from all the sources to all of the sinks as possible.

Extend the flow properties and definitions to the multiple-source, multiple-sink problem. Show that any flow in a multiple-source, multiple-sink flow network corresponds to a flow of identical value in the single-source, single-sink network obtained by adding a supersource and a supersink, and vice versa.

(MM6) **An Automaton Pattern-Matching Example**

Set pattern $P = abcab$.

- Construct the transition function $\delta(q, s)$ for the string matching automaton corresponding to P.
- Draw M. (It is not necessary to show the transitions to state 0.)
- Run M on the text T below. For each character identify the state that M will be in after reading the character.

$$T = a\ c\ a\ b\ c\ a\ b\ c\ a\ b\ a\ b\ c\ a\ b\ c\ a\ b\ c\ a\ b\ a.$$

- Show how this identifies all occurrences of P in T.

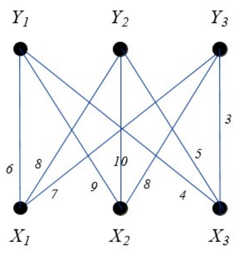
(MM*1) **The Taxi Problem**

Consider a taxi company that has received many reservations. A reservation specifies when and where a taxi needs to be to pick up a passenger and when and where the taxi will drop the passenger off.

The company wants to calculate the minimum number of taxis it will need to service all of those requests. How can it do this?

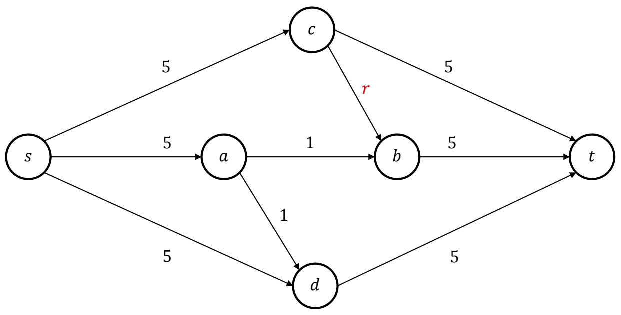
More specifically, you are given n taxi reservations $r_1, r_2, \ldots, r_n$. For every pair of reservations r_i, r_j you are told if the same taxi can first satisfy reservation r_i and then go on to satisfy reservation r_j. Find the minimum number of taxis needed to satsify all of the reservations.

(MM*2) **The Hungarian Algorithm**
Find a max-weight perfect matching in the weighted bipartite graph below:



(MM*3) **Pathological Example of Ford-Fulkerson**

Consider the flow graph



where r is the reciprocal of the Golden Ratio,

$$r = \left(\frac{\sqrt{5}+1}{2} \right)^{-1} = \frac{\sqrt{5}-1}{2} \approx 0.619 \ldots .$$

This has a maximum flow with $|f| = 11$.

Show that there exists an infinite sequence of augmenting path steps whose sum converges to a flow with value less than 11.

Hashing

(HA1) **Open Addressing**
Let table size be $m = 15$ (with items indexed from $0 \ldots 14$).
Use the hash function $h(x) = (x \bmod 15)$ and linear hashing to hash the items $19, 6, 18, 34, 25, 34$ in that order.
Draw the resulting table.

(HA2) **Universal Hashing**
Recall the universal hash function family defined by

$$h_{a,b}(x) = \Big((ax + b) \bmod p\Big) \bmod m$$

where $a \in Z_p^*$, $b \in Z_p$ and p is a prime with $p \geq U$. Let $p = 17$, $m = 5$. For all $x = 0, 1, \ldots, 16$ write the values for $h_{1,0}(x)$. Now write all the values for $h_{2,2}(x)$.