## ❖ Priority Queues and Heapsort

An important data structure that we will see later in this course is a priority queue. Unlike a regular queue where we can only insert elements to the end of the queue, in a priority queue we can insert items we different "priority" levels.

A priority queue

- is an ADT,
- is a collection of items, where
- each item is labeled with a "priority", and
- supports the following operations:
  - Insert item with given priority
  - Delete an item
  - Select the item with the "most urgent" priority.

The "most urgent" could be defined as the largest priority or the smallest priority depending on the application.

There are many ways to implement a priority queue. Let's look at the binary heap implementation. A maxheap is a binary tree where every parent is larger than its two children.

**Question 53.** Draw a maxheap that stores the following elements $S = \{4, 1, 2, 7, 9, 3\}$.

We usually store binary heaps in arrays that represent a binary tree. We use the following rules to represent the tree using our array $H$.

- The root is placed in $H[0]$.
- The left child of $H[i]$ is placed in $H[2i + 1]$.
- The right child of $H[i]$ is placed in $H[2i + 2]$.
- The parent of $H[i]$ is placed in $H[\lfloor (i - 1)/2 \rfloor]$.

**Question 54.** The following is an example of a binary heap. Draw the tree that it is representing.

| 92 | 76 | 81 | 62 | 23 | 79 | 22 | 55 |
|---|---|---|---|---|---|---|---|

Our heap should support at least the following four operations.

- `FindMax(H)`: Find maximum value in heap $H$.
- `ExtractMax(H)`: Find and delete the maximum value from heap $H$.
- `Insert(H, x)`: Insert an item with value $x$ in $H$.
- `Delete(H, i)`: Delete the item at location $i$ from $H$.

To implement most of these, we need some way to generically position items in the correct locations in the heaps. The following two functions will help accomplish that for us.

- `SiftUp(H, i)`: Repeatedly swap the item at position $i$ with its parent until it is in the correct spot.
- `SiftDown(H, i)`: Repeatedly swap the item at position $i$ with its parent until it is in the correct spot.

**Question 55.** For a binary heap with size $n$, what are the maximum number of `SiftUp` or `SiftDown` operations we would need to do?

### 5.0.1 Recursive Heapify

One way we can construct a heap is by the following algorithm. Suppose we have a list of items in an array (which is not heap-ordered), and we want to reorder them so that they represent a maxheap.

```
H = # Unheapified array
def recursive_heapify(k):
    if (2k + 1 <= n):
        recursive_heapify(2k + 1)
    if (2k + 2 <= n):
        recursive_heapify(2k + 2)
    SiftDown(H, k)
```

**Question 56.** What is the runtime of `recursive_heapify` when called with $k = 0$?

### 5.0.2 Extract Max

The other piece we will need is the `extract_max` function. To accomplish this, what we will do is to swap the element at the root with the last element in the heap, and then sift down the new root until we have a valid heap again.

| 92 | 76 | 81 | 62 | 23 | 79 | 22 | 55 |
|---|---|---|---|---|---|---|---|

To wrap things up, our algorithm will first heapify the list of elements we have, then call extract max for every element in the heap until we have deleted everything. Once this is done, our array will be fully sorted.

**Question 57.** What is the total runtime of heapsort?

## 5.1 Linear Time Sorting

We've seen that comparison based sorting (which is the most general way to sort a set of items) requires at least $\Omega(n \log n)$ comparisons. It turns out that if we make some assumptions on the input, we can find scenarios where we can do better!

One such algorithm is **counting sort**. In this algorithm, we assume that our inputs take integer values in some fixed range 1 to $k$. Given this assumption, the total runtime of the algorithm comes out to $\Theta(n + k)$, which for the special case of $k = O(n)$ gives a $\Theta(n)$ time sorting algorithm. The following is a brief description for how the algorithm works.

1. Create new arrays $B$ with length $n$ and $C$ with length $k$.
2. Go through $A$ and use $C$ to store the number of appearances of each element.
3. Go through $C$ to create a cumulative sum of values less than or equal to $i$.
4. Use $C$ to start copying $A$ to $B$ in sorted order.

# ❖ Dynamic Programming

## 6.1 The Scheduling Problem

At Anime Expo, there are a lot of exciting events happening all day. To maximize your time at the event, you want to plan ahead and come up with a schedule that will maximize your happiness. This is an instance of the weighted scheduling problem, which is formally defined below.

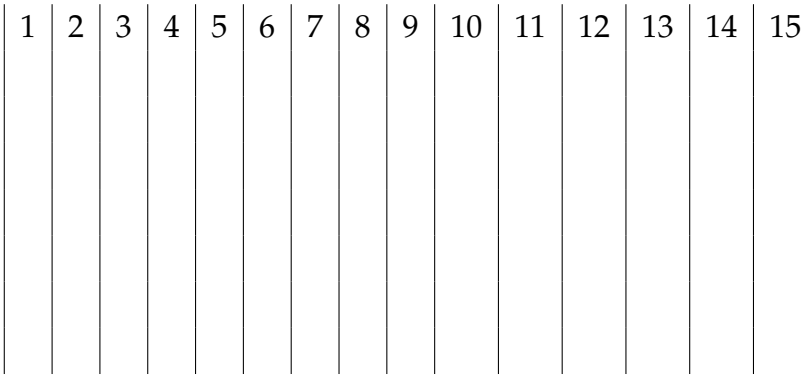
**Problem** (WEIGHTEDSCHEDULING)**.**

**Input:** Collection of $n$ triples in the form $(s(i), f(i), v(i))$, where $s(i)$ is the start time, $f(i)$ is the finish time, and $v(i)$ is the value of the $i$-th activity.

**Output:** A set of non-overlapping events that maximize the total value.

**Example 6.1.** Consider the following schedule.

| $i$ | $s(i)$ | $f(i)$ | $v(i)$ | $p(i)$ |
|---|---|---|---|---|
| 1 | 1 | 3 | 2 | |
| 2 | 2 | 6 | 4 | |
| 3 | 5 | 7 | 4 | |
| 4 | 4 | 10 | 7 | |
| 5 | 8 | 11 | 2 | |
| 6 | 9 | 12 | 1 | |
| 7 | 11 | 15 | 3 | |




To solve this, we will do two things.

1. Sort events by their finishing time (already done for the above example).
2. Compute $p(i)$, which is defined to be the index of the highest interval that does not overlap with $i$, or 0 if that does not exist.

**Question 58.** Fill in the column $p(i)$ in the example above.

**Question 59.** How many total combinations of events are there (including combinations where events overlap)?

**Question 60.** Suppose that for each event $i$ before the last event $n$, someone could tell you what the maximum value and sets were for all the events from 1 through $i$. Their answers are outlined in the table below. Using this information, come up with the maximum value and the sets that form that answer.

| $i$ | $s(i)$ | $f(i)$ | $v(i)$ | $p(i)$ | Maximum Value | Sets |
|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 2 | | 2 | 1 |
| 2 | 2 | 6 | 4 | | 4 | 2 |
| 3 | 5 | 7 | 4 | | 6 | 1, 3 |
| 4 | 4 | 10 | 7 | | 9 | 1, 4 |
| 5 | 8 | 11 | 2 | | 9 | 1, 4 |
| 6 | 9 | 13 | 6 | | 10 | 1, 3, 9 |
| 7 | 12 | 15 | 3 | | ? | ? |

This is great, but in practice we don't have access to these answers. How can we construct them by ourselves? Well, we can start from the smallest problem and slowly build it up using the strategy from above! Let's ignore the sets for now, and just compute the maximum value.

```
def best_value():
    MAXVAL = [0]
    for i in range(1, n+1):
        if v[i] + MAXVAL[p[i]] > MAXVAL[i - 1]:
            MAXVAL.append(v[i] + MAXVAL[p[i]])
        else:
            MAXVAL.append(MAXVAL[i - 1])
    return MAXVAL[-1]
```

Of course, having just the value is not very useful if we don't know the combination of events that can achieve that! To keep track of the events to keep as well, we will create another array keep, that will store True only if we decide that keeping that event maximizes the total value (up to that event).

```
def scheduling():
    MAXVAL = [0]
    keep = [False]  # Add an element so that we can use 1 indexing
    for i in range(1, n+1):
        if v[i] + MAXVAL[p[i]] > MAXVAL[i - 1]:
            MAXVAL.append(v[i] + MAXVAL[p[i]])
            keep.append(True)
        else:
            MAXVAL.append(MAXVAL[i - 1])
            keep.append(False)
```

**Question 61.** Write down what the contents of keep will be if we run this scheduling function for the following instance.

| $i$ | $s(i)$ | $f(i)$ | $v(i)$ | $p(i)$ | Maximum Value | keep |
|---|---|---|---|---|---|---|
| 1 | 1 | 3 | 2 | | 2 | |
| 2 | 2 | 6 | 4 | | 4 | |
| 3 | 5 | 7 | 4 | | 6 | |
| 4 | 4 | 10 | 7 | | 9 | |
| 5 | 8 | 11 | 2 | | 9 | |
| 6 | 9 | 13 | 6 | | 10 | |
| 7 | 12 | 15 | 3 | | | |

**Question 62.** What is the runtime of `scheduling`?

**Question 63.** How can we use the keep array to reconstruct the full solution?

The following is some sample code for printing out the full solution.

```
def print_solution(j):
    if j == 0
        return
    if keep[j]:
        print_solution(p[j])
        print(j)
    else:
        print_solution(j - 1)
```

**Question 64.** What can we say about the running time for `print_solution`?

## 6.2 Rod Cutting

You are working for a company that sells steel rods. You have a factory that can manufacture rods that have a fixed length $n$, and you want to find a way to optimally slice up the rods to maximize your profit. You have a data scientist on your team who analyzes the current market trends to set an optimal price for each length of a rod. Given this, come up with an algorithm to design

**Problem** (RODCUTTING)**.**

**Input:** A rod with length $n$ and an array of length $n$ describing the price of a rod with length $i$ for $1 \leq i \leq n$.

**Output:** The maximum value $r_n$ of the rod achievable by slicing it into pieces with integer lengths, and the slicing that achieves this price.

**Example 6.2.** Suppose we have rods that are manufactured to have length 4. We have the following pricing chart for prices.

| Rod length ($i$) | 1 | 2 | 3 | 4 |
|---|---|---|---|---|
| Price in dollars ($p_i$) | 2 | 5 | 7 | 7 |

**Question 65.** What is the optimal way to slice up our rod to maximize profits?

**Question 66.** If given a rod of length $n$, how many different ways can we slice up the rod?

For the scheduling problem, when we assumed that someone could tell us the answer to all the smaller problems, we could isolate our analysis to deciding whether we should include or leave out a particular task. We want to use a similar line of reasoning.

**Question 67.** If someone could tell us the solution to smaller instances of the rod cutting problem, what's a way that we can use this information without having to check an exponential number of cases?

**Question 68.** Draw the four different first slices that can happen in Example 6.2.

For every problem that can be solved using dynamic programming, there are two ways to solve them. The first is the **top-down approach**. This is fancy language for using a recursive algorithm, where in your first function call you input the full problem, and then make recursive calls to smaller pieces of the array.

```
def cut_rod(p, n):
    if n == 0:
        return 0
    q = -100
    for i in range(1, n + 1):
        q = max(q, p[i] + cut_rod(p, n - i))
    return q
```

I highly encourage you to try running this function on your computer, and you will find that as n increases, even at around 40 it will take a really long time to complete. Why is it so inefficient?

**Question 69.** Draw a tree representing the number and size of recursive calls made at each level for an input with $n = 4$.

### 6.2.1 Top-down Approach

The clever solution of top-down dynamic programming comes from recognizing that many of the subproblems are being solved many times. If we just save the results of subproblems when we compute, then reuse them when we need them again, we can save on a lot of the repetitive computation! We can accomplish this by having an array outside of our function called our "memo", where we store the answers to subproblems. This process is called **memoization**.

```
memo = [0, -100, ..., -100]  # Array with length n + 1
def memoized-cut-rod(p, n):
    if memo[n] >= 0:
        return memo[n]
    q = -100
    for i in range(1, n + 1):
        q = max(q, p[i] + memoized_cut_rod(p, n - i))
    memo[n] = q  # Save the result of n in the memo
    return q
```

### 6.2.2 Bottom-up Approach

If you want to avoid the recursive nature of the top-down approach, the other approach is the **bottom-up approach**. This is how we solved the weighted scheduling problem, where we start with the smallest instance of the problem and use the answers from this size to solve gradually bigger problems.

```
def memoized-cut-rod(p, n):
    memo = [0, -100, ..., -100]  # Array with length n + 1
    for i in range(1, n + 1):
        q = -100
        for j in range(1, i + 1):
            q = max(q, p[j] + memo[i - j])
        memo[i] = q  # Save the result of n in the memo
    return memo[n]
```

**Question 70.** What is the runtime of the bottom up rod cutting algorithm?

### 6.2.3 Bottom-up Approach with Solution

Our algorithms above are only able to tell us the best possible price we can achieve, but not the actual way to cut our rods. Just like the scheduling problem, we need to make sure to add one more array to keep track of which cut produced the best cut.

```
def memoized-cut-rod(p, n):
    memo  = [0, -100, ..., -100]  # Array with length n + 1
    cut = []  # Array with length n
    for i in range(1, n + 1):
        q = -100
        for j in range(1, i + 1):
            q = max(q, p[j] + memo[i - j])
            cut[i] = j  # The best cut location so far for length i rod
        memo[i] = q  # Save the result of n in the memo
    return memo, cut

memo, cut = memoized-cut-rod(p, n)
while n > 0:
    print(cut[n])
    n = n - cut[n]
```