Programming Paradigms for Computation

Solving some Problems

Eg 1: Finding the largest number from n distinct integers.

(a) **Find maximum of n elements.**

```
maximum(A, n) {
  max = A[1]
  for i = 2 to n:
    if max < A[i]:
      max = A[i]
  return max
}
```

Pseudocode 1

Correctness Proof

To prove the algorithm's correctness, we need to establish the *invariants* in our algorithm.

(a) If $n = 1$, our algorithm never goes into the `for` loop, and (correctly) returns `A[1]`.
(b) Inside an iteration of the loop, before the `if` block, `max` stores the value of the maximum of previously processed $i - 1$ elements.
(c) After the `if` block, `max` stores the value of the maximum of the i elements processed.

This can be proved by a simple inductive setup:

Assume that at the kth iteration of the loop, max holds the maximum value of the elements $A[1], A[2], ..., A[k]$.

During the $(k + 1)$th iteration, the algorithm compares `max` with $A[k + 1]$:

- If `max < A[k+1]`, then `max` gets updated to `A[k+1]`, which is indeed the maximum of $A[1], A[2], ..., A[k + 1]$.
- Otherwise, `max` remains unchanged, still retaining the maximum value in $A[1], A[2], ..., A[k + 1]$.

Thus, the loop invariant is maintained.

Algorithm Efficiency

The algorithm makes $n - 1$ comparisons, and in the worst case $(n - 1) + 1 = n$ assignments to `max`.

Let us use the number of comparisons as a way to gauge the time complexity of an algorithm.

Can we show that $n-1$ comparisons is the most optimal in solving this problem?

Notice that each comparison eliminates one candidate for being the maximum. To reduce n candidates to 1 maximum, we need exactly $n-1$ comparisons. So, it is the lower bound for comparisons for an algorithm.

Example: Let the array be $[4, 5, 8, 1, 6, 3, 2, 7]$.

Comparisons in Algorithm 1 can be shown by the following tree diagram.

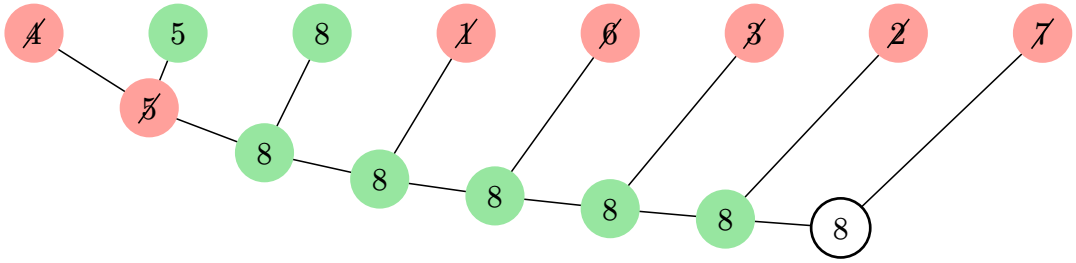

FIGURE 2: Sequential comparisons

There are other ways to find the maximum, such as by doing the comparison in pairs.

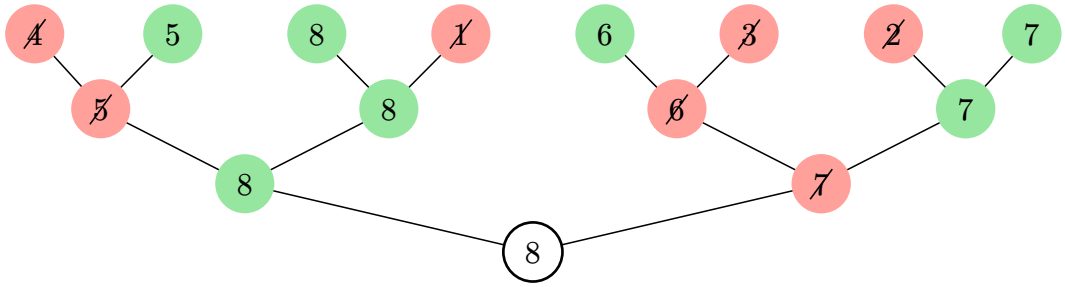

FIGURE 3: Pairwise comparisons

In any such elimination-style algorithm, we will need $n-1$ comparisons to find the maximum.

(b) **Find the largest and smallest number of n elements**.

We can have the same algorithm, which also calculates the minimum along the way:

```
maxmin(A, n) {
  max = A[1]
  min = A[1]
  for i = 2 to n:
    if max < A[i]:  max = A[i]
    if min > A[i]:  min = A[i]
  return max, min
}
```

PSEUDOCODE 4

and its correctness can be proved by a similar setup.

Algorithm 2 takes $2(n-1)$ comparisons to find the maximum and minimum values, essentially solving both the problems separately ($n-1$ comparisons for each). Can we do any better?

We can use the calculations we are doing for finding the maximum to our advantage. In particular, we can observe that the smallest number will be out of the ones which has been eliminated in the first round while finding the maximum!

What this means is instead of finding the minimum of n elements, we now need to find minimum of m elements, where m is the number of elements which got eliminated in the first round. Then, the total number of comparisons for this algorithm would be $n-1+m-1$.

Clearly, we want to find minimum possible value for m. How can we minimize the number of elements that get eliminated in the first round? Equivalently, we need to minimize the number of comparisons happening in the first round.

Clearly, this is possible if we pair up the elements and have $\frac{n}{2}$ comparisons in the first round, as shown in Figure 2. By doing so, only half of the elements (that got eliminated in the first round) are the ones which have to be checked to find the minimum.

So, the optimal number of comparisons is $n-1+n/2-1 = 3n/2-1$.

(c) **Find the largest, and second largest element of n elements**.

Continuing like previously, we need to limit the search space for finding the second largest element, to find a more efficient algorithm that $2(n-1)$ comparisons.

The key observation here is that the second largest element will be present in the set of elements which lost to the winner (maximum element) during the eliminations!

Here's the justification: Let the largest element be a_k and the second largest element be a_l.

- Clearly, a_l has been eliminated at least once (otherwise, it would have been the winner).
- The only matchup to which a_l loses is against a_k itself. So it is necessary that there was one comparison between a_l and a_k, in which a_l got eliminated.

Now, we know that the second largest element is present in the set of elements that get eliminated against the maximum element. To minimize the size of this set, we want to minimize the height of the decision tree, which once again occurs when we do binary tree-style pairings.

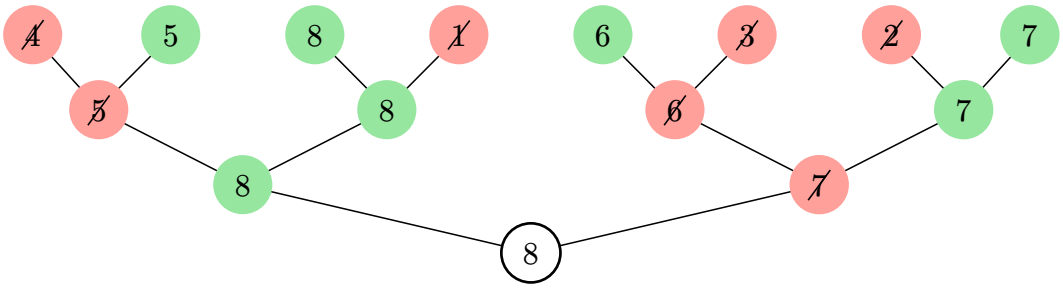

FIGURE 5: Pairwise comparisons

The number of elements that face up against the maximum element in the tournament is $\log_2 n$.

Thus, the most optimal algorithm has $n - 1 + \log_2(n) - 1 = n + \log_2(n) - 2$ comparisons, to find both the largest and the second largest element.

The above method can be extended to sort any array. In particular, we can find the k^{th} largest element in at most $\log_2 n$ comparisons, so the array can be sorted in $n \log_2 n$ comparisons.

Heap sort is such an algorithm.

- Array representation of binary tree is done by taking a_1 as the root node, and a_{2n} and a_{2n+1} as left and right child nodes of a_n.
- A complete binary tree is one which has all its nodes filled except the last level, where elements must be filled from left to right.
- A max (min) heap is a complete binary tree such that value of every node is greater (smaller) than all its descendants.
- Inserting into a max heap:
  - Add the element as a leaf node.
  - Continually move the element up if it is greater than its parent.

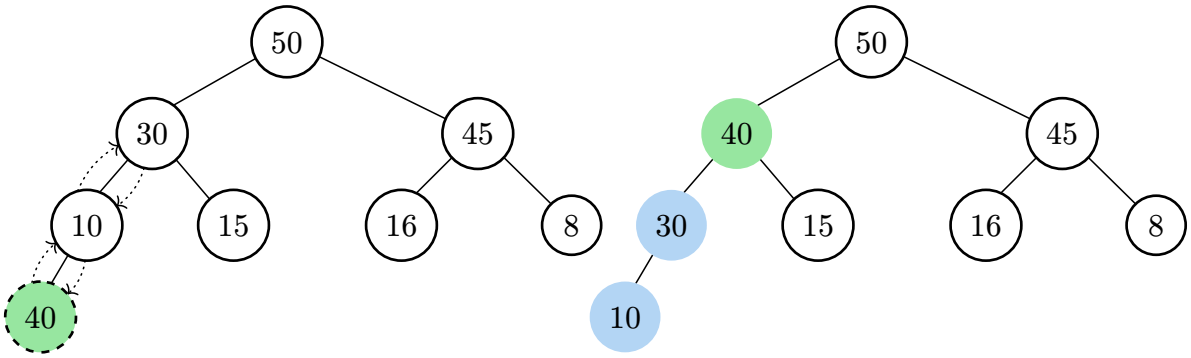

FIGURE 8: Insertion into a max heap

Requires $\lfloor \log_2(n+1) \rfloor$ operations in the worst case.

- Deleting the maximum element from a max heap:
  - Remove the element and replace with a leaf node
  - Continually move the element down if it is smaller than its child (choose the larger child)

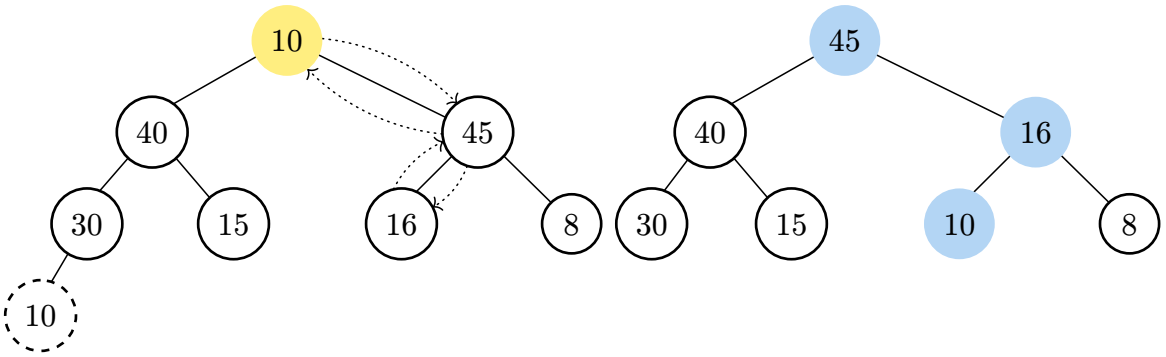

FIGURE 11: Removal of max element from a max heap

Requires $\lfloor \log_2(n-1) \rfloor$ operations in the worst case.

By repeatedly removing the max element, we can sort the array in $n \log n$ time complexity.

Eg 2: Find $F(n)$ for $n > 2$, if $F(n) = F(n-1) + F(n-2)$, $F(0) = 0$, $F(1) = 1$.

The Fibonacci series can be generated by an algorithm. Due to its recursive definition, we can define it recursively, as follows:

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

PSEUDOCODE 12

However, the issue with this algorithm is that it does a lot of redundant computations.

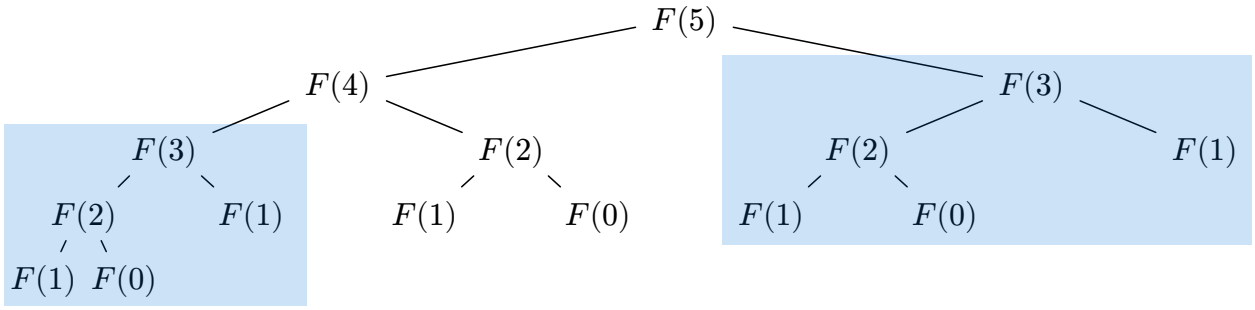

FIGURE 13: Recursive implementation of Fibonacci Series

As we can see, many of the calculations are computed more than once, for instance, we are having to calculate $F(3)$ twice in the recursion to find $F(5)$.

How many addition operations do we perform? That is given by the following recurrence relation:

$$\begin{aligned} T_n &= 1 + T_{n-1} + T_{n-2} \\ T_0 &= T_1 = 0 \\ \Rightarrow T_n &= F(n) - 1 \end{aligned}$$

To remove the redundancies we can use an iterative solution, which does not branch out recursively, preventing this issue.

```
fib(n) {
  if n <= 1: return n
  a, b = 0, 1
  for i = 2 to n {
     a, b = b, a + b
  }
  return b
}
```

Pseudocode 14

This has a linear complexity, as only $n-2$ additions take place.

However, not all such problems have a nice iterative solution. Consider:

$$R(n) = \begin{cases} R(2n+2) + R\left(\frac{n}{2}\right) & \text{if } n \text{ is odd} \\ R\left(\frac{n}{3} - 4\right) + R\left(\frac{n}{10} - 12\right) & \text{if } n \text{ is even} \\ 1 & \text{if } n \leq 0 \\ 10 & \text{if } n \geq 10^8 \end{cases}$$

To solve this iteratively would mean to find the values of $R(k)$ for all $k = 1, 2, ..., n$, which doesn't seem to be a great idea.

How can we solve the problem of redundant computations in the recursive solution? The answer is to *remember past solutions*.

```
results = []
R(n) {
  if n <= 0:     return 0
  if n >= 1e8:   return 10
  if results[n]: return results[n]
  if n % 2 == 1 {
    ans = R(2*n + 2) + R(n/2)
  } else {
    ans = R(n/3 - 4) + R(n/10 - 12)
  }
  results[n] = ans
  return ans
}
```

Pseudocode 15: Memoization

This ensures we do not compute the value of R at the same n twice. This technique is usually termed as **Dynamic Programming**.

There is one final issue, which is that there may be cyclic dependencies in the definition of R. That is, it is possible that the following happens:

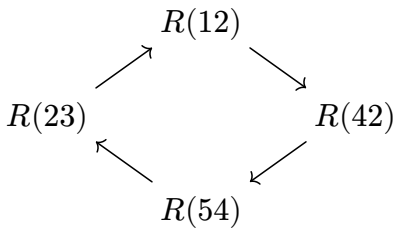

FIGURE 16: Cyclic Dependency

To detect such a cyclic dependencies, we need to store some more information, for each n, which of the following state we are in:
(a) never encountered
(b) encountered and entered into recursion
(c) encountered and computed (value is available)

Now, at the beginning of $R(n)$ we can check which state we are in, and if we are in state (b), we can report a cyclic dependency.

Eg 3: Multiply two n-bit binary numbers.

$$\begin{array}{rcccccccc} & & & & 1 & 0 & 1 & 0 \\ \times & & & & 1 & 1 & 0 & 1 \\ \hline & & & & 1 & 0 & 1 & 0 \\ & & & 0 & 0 & 0 & 0 & \\ & & 1 & 0 & 1 & 0 & & \\ & 1 & 0 & 1 & 0 & & & \\ \hline 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{array}$$

FIGURE 17: Long Multiplication for n-bit binary numbers

The above is naive multiplication, which has a time complexity of $O(n^2)$. It can be implemented by shifting and adding binary numbers appropriately.

Count number of occurences of substring in given string

Given two strings P and Q, find number of occurences of Q in P including overlaps.

The naive solution uses a total of $m(n-m+1)$ character comparisons in the worst case.

KMP (Knuth-Morris-Pratt) Algorithm

1. Define $\pi(s)$ (prefix function / longest prefix-suffix) where $\pi[i]$ = longest proper prefix of $s[:i]$ which is also a suffix.

a	b	c	a	b	d	a	b	c	d
0	0	0	1	2	0	1	2	3	0

FIGURE 18: Prefix array

a	a	a	a	a	a	a
0	1	2	3	4	5	6

2. Consider the string Q#P. Find its prefix function.

p	y	#	p	y	t	h	o	n
0	0	0	1	2	0	0	0	0

a	b	a	b	#	a	b	c	d	a	b	a	b	a	b
0	0	1	2	0	1	2	0	0	1	2	3	4	3	4

   Observe, that the number of occurences of m (length of Q) in the prefix array in the last n positions, gives the number of occurences of the substring Q in P. If we can compute $\pi(s)$ in $O(n)$, then this problem can be solved in $O(m+n)$.

Alternatively, compute $\pi(Q)$. Iterate over P and Q using 2 pointers, incrementing i and moving j to $\pi[j]$ if they don't match, and incrementing both if they do.

Computing $\pi(s)$

1. Set $\pi[0] = 0$.
2. For some $i > 0$, set $j = \pi[i-1]$.
3. Check whether $s[i] = s[j]$. If so, set $\pi[i] = j+1$, continue for next i. Otherwise, set $j = \pi[j-1]$ and repeat.
4. If $j = 0$, set $\pi[i] = 0$.

```
def prefix(s):
  pi = [0] * len(s)
  for i in range(1, len(s)):
    j = pi[i-1]
    while j > 0 and s[i] != s[j]: j = pi[j-1]
    if s[i] == s[j]: j += 1
    pi[i] = j
  return pi
```

Proofs

1. **Insertion sort can be suitably developed to work in $O(n \log n)$ time.**

   What is insertion sort?

2. No.

   Counter example:

   $$S_1 = \text{"pqarbsct"}$$
   $$S_2 = \text{"abcpqrst"}$$
   $$S_3 = \text{"lmnopabc"}$$

   Here, $\text{LCS}(S_1, S_2) = \text{"pqrst"}$ and $\text{LCS}(\text{"pqrst"}, S_3) = \text{"p"}$ which is not $\text{LCS}(S_1, S_2, S_3) = \text{"abc"}$