## 3.1 The divide-and-conquer approach

Many useful algorithms are recursive in structure: to solve a given problem, they call themselves recursively one or more times to deal with closely related sub-problems.

A quick example of a recursion can be finding a factorial of a positive integers, $n$.

$$n! = n * (n-1)!\ \forall n \in \mathbb{Z}^+ \text{ where } 0! = 1$$

$n! = 5 * 4! = 5 * 4 * 3!... = 5 * 4 * 3 * 2 * 1$

Although finding a factorial of a positive integer has a recursive structure, it may not be a very good example for divide-and-conquer approach.

The divide-and-conquer primarily involves three steps at each level of the recursion:

- **Divide** the problem into a number of sub-problems that are smaller instances of the same problem.
- **Conquer** the sub-problems by solving them recursively. If the sub-problem sizes are small enough, however, just solve the sub-problems in a straightforward manner.
- **Combine** the solutions to the sub-problems into the solution for the original problem.

## 3.2 Exponentiation by squaring

If we have to calculate $f(n) = 2^n$ for a given value of $n$, a straightforward approach will be to iterate over $\langle 1, 2, 3, ..., n\rangle$, to build $n$ multiples of 2.

```
ans = 1
for i = 1 to n:
    ans = ans * 2
return ans
```

The for loop runs for $n + 1$ times with $O(1)$ for each iteration. Therefore, it will have a linear runtime, $O(n)$.

Can we do better than this?

Let's rewrite the same problem in a recursive structure:

$$f(n) = \begin{cases} 0 & n = 1 \\ (f(n/2))^2 & \text{if n is even} \\ 2 * (f((n-1)/2))^2 & \text{if n is odd} \end{cases}$$

To evaluate $2^8$, $n = 8$, the function calls we need are $f(4), f(2), f(1), f(0)$. These function calls are enough to build the solution for $f(8)$.

$f(8) = (f(4))^2$
$f(4) = (f(2))^2$
$f(2) = (f(1))^2$
$f(1) = (f(0))^2$

$f(0) = 1$

What will be the function calls for $n = 11$? You can write out the steps for $2^{11}$, where $n = 11$ is odd.

The pseduocode for this will be:

```
Exponentiator(n):
    if n = 0:
        return 1
    else if n mod 2 = 0: \\ n is even
        x = Exponentiator(n/2)
        return x * x
    else: \\ n is odd
        x = Exponentiator((n-1)/2)
        return 2 * x * x
```

Please note that the we are only using one function call at each recursion level. If the pseudocode was "return Exponentiator(n/2)*Exponentiator(n/2)" we will be doing half redundant recursive calls, making the performance much worse.

### 3.2.1 Proof of correctness

We can use proof by induction to show the correctness of this algorithm.

**Base case:** When $n = 0$, the algorithm returns 1, which is true, because $2^0 = 1$. Therefore, base case holds.

**Inductive Step:** Let's assume that Exponentiator(k) gives us the right output, $2^k$, for $k < n$, $2^k$. Then, we have to show that Exponentiator(n) gives us the right output, which is $2^n$.

Case 1: If $n$ mod $2 = 0$ or $n$ is even, the algorithm returns $x^2$, where $x =$ Exponentiator($n/2$). From the inductive hypothesis, we know that Exponentiator($k$) gives us the right output. In case of even value of n, $k = n/2$ for the subproblem. As per the inductive hypothesis above, $x = 2^{n/2}$. Therefore, $x^2 = (2^{n/2})^2 = 2^n$, which is the expected output.

Case 2: If $n$ is odd, the algorithm returns $2x^2$, where $x =$ Exponentiator((n-1)/2). Using the inductive hypothesis, for $k = (n-1)/2 < n$, $2 * x^{(n-1)/2} = 2^n$.

In either case, we found that Exponentitor($n$) is giving us the right output, $2^n$ for $n$. Thus, we show that the algorithm sketched out above gives us the correct answer for $n \geq 0$.

*Note: This proof of inductive will not hold if n is a real number.*

### 3.2.2 Runtime analysis

The runtime, $T(n)$, for Exponentiator can be expressed in recurrence relation:

$$T(n) = \begin{cases} c_1 & \text{if } n = 1 \\ T(n/2) + c_2 & \text{if n is even} \\ T((n-1)/2) + c_3 & \text{if n is odd} \end{cases}$$

We will work on evaluating the runtime of recurrence relation in the next lecture. However, you can guess that the runtime should be better than $O(n)$ because at each recursive call, we are working with $n/2$ input size. If we keep dividing the input size by 2, how many iterations will it take to hit the base case $n = 0$?

## 3.3 Merge Sort

Previously, we looked at insertion sort, which has a runtime in quadratic order, $an^2 + bn + c$ for the input array size, $n$. We can express the runtime of insertion sort as $\Theta(n^2)$. We can improve this runtime to $\Theta(n \lg n)$ to sort an array of size using merge sort.

We can roughly describe ***merge sort*** in a divide-and-conquer approach as follows:

- **Divide**: Divide the $n$-element sequence to be sorted into two subsequences of $n/2$ elements each.
- **Conquer** Sort the two subsequences recursively using merge sort.
- **Combine** Merge the two sorted subsequences to produce the sorted answer.

Example: If we are using merge sort to sort array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$, we split the array into two halves: $\langle 5, 2, 4, 7 \rangle$ and $\langle 1, 3, 2, 6 \rangle$. We keep dividing each subarray till the length of the subarray is 1.

1. $\langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$
2. $\langle 5, 2, 4, 7 \rangle$, $\langle 1, 3, 2, 6 \rangle$
3. $\langle 5, 2 \rangle$, $\langle 4, 7 \rangle$, $\langle 1, 3 \rangle$, $\langle 2, 6 \rangle$
4. $\langle 5 \rangle$, $\langle 2 \rangle$, $\langle 4 \rangle$, $\langle 7 \rangle$, $\langle 1 \rangle$, $\langle 3 \rangle$, $\langle 2 \rangle$, $\langle 6 \rangle$

Once the recursion "bottoms out" when the length of the subarray is 1, we start merging the **sorted** subarrays. At the base case, each subarray is sorted trivially, for their size is 1.

Once the recursion "bottoms out", Merge procedure starts kicking in, as the stack of recursive functions calls starts getting cleared, and called out.

5. $\langle 2, 5 \rangle$, $\langle 4, 7 \rangle$, $\langle 1, 3 \rangle$, $\langle 2, 6 \rangle$
6. $\langle 2, 4, 5, 7 \rangle$, $\langle 1, 2, 3, 6 \rangle$
7. $\langle 1, 2, 2, 3, 4, 5, 6, 7 \rangle$

The key operation in the merge sort algorithm is merging two sorted sequences in the "combine" step. We merge by calling an auxiliary procedure Merge(A, p, q, r), where $A$ is an array to be sorted, $p$, $q$, and $r$ are the indices of $A$, such that $p \leq q < r$. The Merge procedure assumes that the subarrays $A(p, q)$ and $A(q + 1, r)$ are already sorted. It **merges** them to form a single sorted subarray that replaces the current subarray $A[p, ..., r]$.

**The Merge procedure takes time $\Theta(n)$, where $n = r - p + 1$ is the total number of elements being merged.** This is also known as two-finger algorithm. With two sorted subarrays, we can pick the smallest items out of two subarrays, and copy it to the sorted combined array. Picking the smallest item out of two sorted subarrays takes a constant time.

```
MERGE($A, p, q, r$)
1  $n_1 = q - p + 1$
2  $n_2 = r - q$
3  Let $L[1, ..., n_1 + 1]$ and $R[1, ..., n_2 + 1]$ be new arrays.
4  for $i = 1$ to $n_1$
5      $L[i] = A[p + i - 1]$
6  for $j = 1$ to $n_2$
7      $R[i] = A[q + j]$
8  $L[n_1 + 1] = \infty$
9  $R[n_2 + 1] = \infty$
10 $i = 1$
11 $j = 1$
12 for $k = p$ to $r$
13     if $L[i] \leq R[j]$
14         $A[k] = L[i]$
15         $i = i + 1$
16     else
17         $A[k] = R[j]$
18         $j = j + 1$
```

In the Merge procedure above, the procedure takes array $A$, where subarrays $A[p, q]$, and $A[q + 1, r]$ are sorted. In order to merge these two subarrays, we create two new arrays, $L$ and $R$, where $|L| = q - p + 1$, and $|R| = r - q$. $L$ and $R$, can basically hold one more items than the subarrays they are supposed to copy. We copy the elements of $A[p, q]$ into $L$, and $A[q + 1, r]$ to $R$. We keep a **sentinel** value, $\infty$, at the end of $L$, and $R$. It's handy to keep copying the leftover elements of the larger subarrays, when the smaller array is emptied. Think of merging $\langle 1, 4\rangle$ and $\langle 2, 3, 5, 6, 7, 8\rangle$. As we keep merging, we reach a point where, merged elements are $\langle 1, 2, 3, 4\rangle$, and there is nothing left in the smaller subarray to compare with. However, if $L = \langle 1, 4, \infty\rangle$, and $R = \langle 2, 3, 5, 6, 7, 8, \infty\rangle$, the previous situation will look be $L = \langle\infty\rangle$, and $R = \langle 5, 6, 7, 8, \infty\rangle$. This makes it convenient to keep copying the leftover sorted elements of $R$ to the merged array.

### 3.3.1 Proof of correctness

#### Proof of Merge Procedure

We will write the proof of correctness of the Merge procedure described above. First we need to define the loop invariant in this procedure.

**Lemma 1.** Loop invariant: **At the start of each iteration of the for loop of lines 12-18, the subarray $A[p, ...k-1]$ contains the $k-p$ smallest elements of $L[1, ..., n_1+1]$**

**and** $R[1, ..., n_2 + 1]$ **in sorted order.** Moreover, $L[i]$ and $R[j]$ are the smallest elements of their arrays that have not been copied back into $A$.

We must show that this loop invarant holds prior to the first iteration of the **for** loop of lines 12-18, that each iteration of the loop maintains the invariant, and that invariant provides a useful property to show correctness when the loop terminates.

**Initialization (Base case)**: Before the first iteration, we have not copied anything from $L$ and $R$ to $A$, which shows $A[p, ...k - 1]$ is empty, i.e. trivially sorted. We should note that $i = j = 1$, and both $L[i]$, and $R[j]$ are the smallest elements of their arrays.

**Maintenance (Inductive step)**: Let's assume that $A[p, ..., k - 1]$ contains the $k - p$ smallest elements of $L$ and $R$ in sorted order.
Case 1: When $L[i] \leq R[j]$, then $L[i]$ is the smallest element not yet copied back into $A$. Because $A[p, ..., k - 1]$ contains the $k - p$ smallest elements, after line 14 copies $L[i]$ into $A[k]$, the subarray $A[p, ...k]$ will contain the $k - p + 1$ smallest elements. With the value of $k$ incremented by the **for** loop, and the value of $i$ to $i + 1$ by line 15, reestablishing the loop invariant for the next iteration.
Case 2: When $L[i] > R[j]$, $R[j]$ is copied to $A$, which is the next smallest elements to be copied to $A$ such that $A[p, ..., k+1]$ is sorted with $k-p+1$ smallest elements. Loop invariant is maintained in this case as well.

**Termination**: The Merge procedure stops, when $k = r + 1$. By the loop invariant, the subarray $A[p, ..k - 1]$, which is $A[p..r]$, contains the $k - p = r - p + 1$ smallest elements of $L$ and $R$ in sorted order. The arrays $L$ and $R$ together contain $n_1 + n_2 + 2 = r - p + 3$ elements. All but the two largest have been copied back into $A$, and these two largest elements are the sentinels.

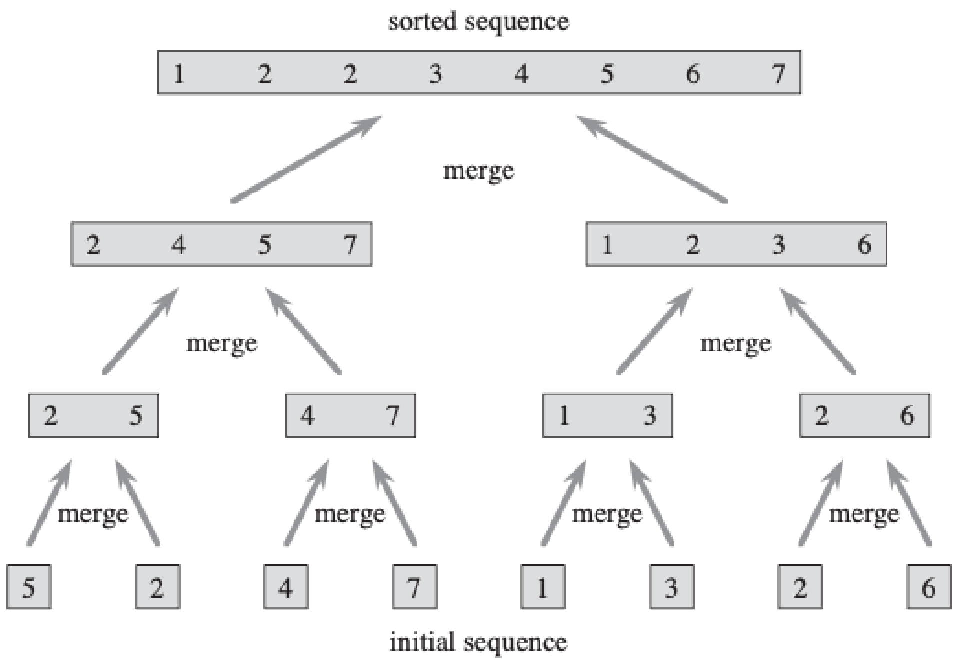

**Figure 2.4** The operation of merge sort on the array $A = \langle 5, 2, 4, 7, 1, 3, 2, 6 \rangle$. The lengths of the sorted sequences being merged increase as the algorithm progresses from bottom to top.

```
MERGE-SORT($A, p, r$)
1  if $p < r$
2      $q = \lfloor (p + r) \rfloor / 2$
3      MERGE-SORT($A, p, q$)
4      MERGE-SORT($A, q + 1, r$)
5      MERGE($A, p, q, r$)
```

## Proof of Merge-Sort Procedure

We will use proof by induction to show the correctness of Merge-Sort procedure. Let's assume that the size of an array to be sorted is $n$.

**Base Case**: When $n = 1$, $A$ is sorted trivially.

**Inductive Step**: Let's assume that Merge-Sort procedure sorts the subarray of size less than $n$. The first half of $A$ will be less than $n$, and so will be the second half. Previously, we showed that Merge procedure sorts two subarrays which are already sorted. If Merge-Sort procedure in line 3 sorts subarray $|A[p...q]| = \lfloor n/2 \rfloor$, and line 4 will sort $|A[q + 1, ...r]| = A.length - \lfloor n/2 \rfloor$, Merge will sort entire array of size $n$ with the Merge procedure in line 5. This concludes the proof.

### 3.3.2 Runtime analysis

Say, we have an array of size $n$, and we divide the array into $n/b$ subarrays. Say, if $n \leq c$, for some constant $c$, and it can be solved with constant set of instructions, we write $T(n) = \Theta(1)$. It takes time $T(n/b)$ to solve one subproblem of size $n/b$, and so it takes $aT(n/b)$ to solve $a$ of them. If we take $D(n)$ time to divide the problem into subproblems and $C(n)$ time to combine the solutions to the subproblems into the solution to the original problem, we get the recurrence:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{otherwise} \end{cases}$$

For the merge sort, $a = 2$, and $b = 2$, because we divide the array into two halves, making $b = 2$, and we merge the two halves making $a = 2$. Since, $D(n)$ and $C(n)$ are both linear functions, we can combine them to $\Theta(n)$ as we are adding two linear functions. This gives us the recurrence relation of Merge-Sort as:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq 1 \\ 2T(n/2) + \Theta(n) & \text{otherwise} \end{cases}$$

## 3.4 Classwork

1. Using Figure 2.4 as a model, illustrate the operation of merge sort on the array $A = \langle 3, 41, 52, 26, 38, 57, 9, 49 \rangle$.

2. Write a recurrence relation for runtime for the following procedure:

```
Power(c, n):
    """
    c and n both are positive integer.
    """
    if n = 1:
        return 1
    else:
        return c * Power(c, n-1)
```

3. Write a recurrence relation for the following procedures:

```
Factorial(n):
    if n < 1:
        return 1
    else:
        return n * Factorial(n-1)
```

4. Write a recurrence relation for the following:

```
Fib(n):
    if n = 0 or n = 1:
         return 1
    else:
         return Fib(n-1) + Fib(n-2)
```

5. Can you think of any other recursive algorithms?

---

*For Monday's class, if we have time, we will go over the following section, otherwise, we will go over these on Wednesday.*

## 3.5 Solving Recurrence by Substitution

The **substitution method** for solving recurrences comprises two steps:

1. Guess the form of the solution.

2. Use mathematical induction to find the constants and show that the solution works.

**Example 1.** What will be the upper bound of $T(n) = T(n-1) + 1$?

Let's guess that $T(n) = O(n)$. In order to show this we need to show $T(n) < cn$ for some $c > 0$, and $n \geq n_0$.

We assume that the bound holds for all positive $m > n$, in particular $m = n-1$, yielding $T(n-1) \leq c(n-1)$.

$$\begin{aligned} T(n) &\leq c(n-1) + c_1 \\ &= cn - c + c_1 \\ &\leq cn \end{aligned}$$

$T(1) = 1$, therefore, for $c \geq 1$, and $n \geq 1$, $T(n) \leq cn$ holds. Therefore, $T(n) = O(n)$.

**Example 2.** Let's find the upper bound on the recurrence: $T(n) = 2T(\lfloor n/2 \rfloor) + n$.

Since we are looking for an upper bound on the recurrence above, we guess that the solution is $T(n) = O(n \lg n)$. The substitution method requires us to prove that $T(n) \leq cn \lg n$ for an appropriate choice of the constant $c > 0$ and $n \geq n_0$. We start assuming that this bound holds for all positive $m < n$, in particular for $m = \lfloor n/2 \rfloor$, yielding $T(\lfloor n/2 \rfloor) \leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n$. Substituting into the recurrence yields

$$\begin{aligned} T(n) &\leq c\lfloor n/2 \rfloor \lg(\lfloor n/2 \rfloor) + n \\ &\leq cn \lg(n/2) + n \\ &= cn \lg n - cn \lg 2 + n \\ &= cn \lg n - cn + n \\ &\leq cn \lg n, \end{aligned}$$

where the last step holds as long as $c \geq 1$.

However, we also need to *choose* $n \geq n_0$. When $n = 1$, $T(n) \leq cn \lg n = c1 \lg 1 = 0$, which is not true, as $T(1) = 1$. Since we get to choose $n_0$ that holds $T(n) \leq cn \lg n$ for $c \geq 1$, we can choose $n > 3$. With $T(1) = 1$, we can derive $T(2) = 4$, and $T(3) = 5$.

When $n = 2$, $T(2) \leq c2 \lg 2$, and when $n = 3$, $T(3) \leq c3 \lg 3$. For $c = 3$, $T(2) \leq 3 * 2 \lg 2 = 6$, $T(3) \leq 3 * 3 \lg 3 = 14.26....$ This fixes our base case irregularity. Therefore, we can make $n = 2$ and $n = 3$ as the base cases for this recurrence.

**Example 3.** Show that $T(n) = T(n-1) + n^2$ is $T(n) = O(n^3)$.

We need to show that $T(n) \leq cn^3$ for some $c > 0$ and $n \geq n_0$.

Let's assume $T(n-1) < c(n-1)^3$.

$$\begin{aligned}
T(n) &\leq c(n-1)^3 + n^2 \\
&= c(n-1)(n-1)^2 + n^2 \\
&\leq c(n-1)n^2 + n^2 \\
&= n^2(cn - c + 1) \\
&\leq n^2(cn) \\
&= cn^3
\end{aligned}$$

$T(1) = 1$ and $T(2) = T(1) + 2^2 = 5$, therefore, for $c \geq 1$, and $n \geq 1$, $T(n) \leq cn^3$. Therefore, $T(n) = O(n^3)$.

### 3.5.1 Making a good guess

Unfortunately, there is no general way to make a good guess. However, we can use recursion trees to make a good guess. If a recurrence is similar to one you have seen before, that can be a really good guess.

For instance, $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n$, we can make a guess that $(\lfloor n/2 \rfloor + 17 \sim \lfloor n/2 \rfloor)$ as $n$ grows. It becomes easier to see $T(n) = 2T(\lfloor n/2 \rfloor + 17) + n \sim 2T(\lfloor n/2 \rfloor) + n$. In that case, our guess can be $T(n) = O(n \lg n)$.

A similar guess can be made about $T(n) = T(n-k) + c_1$ template, or $T(n) = T(n-k) + n^2$.

With some recurrence relations in a bit "simpler" forms, we can enumerate the values of $T(n)$ for some $n$ to find a more generalized solution.

**Example 3.** $T(n) = T(n-1) + c_1$

$$\begin{aligned}
T(n) &= T(n-1) + c_1 \\
&= T(n-2) + c_1 + c_1 \\
&= T(n-3) + c_1 + c_1 + c_1 \\
&= ... \\
&= T(n-k) + k * c_1
\end{aligned}$$

when once we hit the largest possible value of $k$, $k = n$, we get

$$\begin{aligned}
T(n) &= T(n-n) + n * c_1 \\
&= T(0) + n * c_1 \\
&= nc_1
\end{aligned}$$

### 3.5.2 Possible Pitfalls

A wrong guess will give a very wrong answer. For instance, in case of $T(n) = 2T(\rfloor n/2 \lfloor) + n$, if we guess that $T(n) \leq cn$, we will end up with a different asymptotic notation.

$$\begin{aligned} T(n) &\leq 2(c\lfloor n/2 \rfloor) + n \\ &\leq 2 * c * (n/2) + n \\ &= cn + n \\ &= O(n) \Leftarrow \textbf{wrong!!} \end{aligned}$$

### 3.5.3 Changing variables

Sometimes changing variable can be helpful. Here is an example for that.

$$T(n) = 2T(\lfloor \sqrt{n} \rfloor) + \lg n.$$

It's a bit difficult to guess with this one, as it looks nothing like what we have done so far. Is there a way we can make the terms a bit simpler to make a guess?

Let's rename $m = \lg n$. That gives us $n = 2^m$. (This is just "inverting" lg to the right side). Now we re-write the previous equation as:

$$T(2^m) = 2T(\lfloor 2^{m/2} \rfloor) + m$$

To make it even simpler, we can rename $S(m) = T(2^m)$, which yields:

$$S(m) = 2S(m/2) + m$$

which looks much similar to $T(n) = 2T(n/2) + n$, which was $T(n) = O(n \lg n)$.

Therefore, $T(n) = T(2^m) = S(m) = O(m \lg m) = O(\lg n \lg \lg n)$.

So far I have enlisted examples mostly from the CLRS. We will try more examples in the class, and homework.