3: Divide and Conquer (Analyzing Recursive ..): Binary Search, Recurrences, Repeated Substitution Method, Merge sort,

Recursive algorithms and recurrences, Merge sort.

Divide big problems into small problems of the same nature. Then solve the sub-problems recursively until it is so small that you can solve the problem trivially.

factorial n!

- Recursive algorithm

INPUT: n – a non-negative integer.
OUTPUT: fac – a non-negative integer that equals n!

```
FACTORIAL(n)
int fac=1;
if n==1 then fac=1                Trivial case
else fac=n*FACTORIAL(n-1)         Divide & Conquer
return fac
```

- Non-recursive algorithm

INPUT: n – a non-negative integer.
OUTPUT: fac – a non-negative integer that equals n!

```
FACTORIAL(n)
int fac=1;
for i from 1 to n
     fac=fac * i
return fac
```

Divide-and-conquer

Divide-and-conquer method for algorithm design:

- If the problem size is small enough to solve it in a straightforward manner, solve it. Otherwise, do the following:
    - **Divide**: Divide the problem into a number of *disjoint* subproblems.
    - **Conquer**: Use divide-and-conquer recursively to solve the subproblems.
    - **Combine**: Take the solutions to the sub-problems and combine these solutions into a solution for the original problem.

Binary Search

Non-recursive version:

INPUT: A[1..n] – a sorted (non-descending) array of integers, *q* – an integer.
OUTPUT: an index *j* such that A[*j*] = *q; 0*, if ∀*j* (1≤*j*≤*n*): A[*j*] ≠ *q*

```
Left=1
Right=n
do
   j=⌊(left+right)/2⌋
   if A[j]==q then return j
   else if A[j]>q then right=j-1
   else left=j+1
while left<=right
return 0
```

Recursive version:

INPUT: A[1..n] – a sorted (non-descending) array of integers, *q* – an integer, l – an integer, left bound, r – an integer, right bound.
OUTPUT: an index *j* such that A[*j*] = *q; 0*, if ∀*j* (1≤*j*≤*n*): A[*j*] ≠ *q*

```
Binary-search(A, l, r, q):
   if l == r then
      if A[l] == q then return l
      else return 0

   m=⌊(l+r)/2⌋

   if A[m] ≥ q then return Binary-search(A, l, m, q)
   else return Binary-search(A, m+1, r, q)
```

Trivial case

Divide & Conquer

Recurrences

- Running times of algorithms with **recursive calls** can be described using recurrences.
- A **recurrence** is an equation or inequality that describes a function in terms of its value on smaller inputs.
- Assume that
  - If the problem size is small enough, the problem can be solved in constant time, i.e.., **Θ(1)**.
  - The division of problem yields **a** sub-problems and each subproblem is **1/b** the size of the original.
- We have:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n \leq c\,, \\ \underbrace{\boxed{aT(n/b)}}_{\text{Conquer}} + \underbrace{\boxed{D(n)}}_{\text{Divide}} + \underbrace{\boxed{C(n)}}_{\text{Combine}} & \text{otherwise}\,. \end{cases}$$

Recurrence on Binary Search

- a = 1, b=2, having one sub-problem with half elements in the array.
- D(n) = Θ(1), computing the middle index, constant time.
- C(n) = 0, no need to combine.

$$T(n) = \begin{cases} \Theta(1) & if\, n = 1 \\ T(n/2) + \Theta(1) & otherwise, i.e., if\;\; n > 1 \end{cases}$$

The Repeated Substitution Method

- Solving recurrences with the repeated substitution method
  - Substitute
  - Expand
  - Substitute
  - Expand
  - ...
- Observe a *pattern* and write how your expression looks after the i-th substitution.
- Find out what the value of i should be to get the base case of the recurrence T(1).
- Insert the value of T(1) and the expression of i into your expression.

Example

e, f are constants.
e: cost for solving a trivial case.
f: cost for dividing

- T(n)=
  - e if n=1
  - T(n/2) + f otherwise, i.e., if n>1
- T(n)= T(n/2) + f
  - =(T(n/4)+f)+f = T(n/4) + 2*f
  - =(T(n/8)+f)+2*f = T(n/8) + 3*f
  - =(T(n/16)+f)+3*f = T(n/16) + 4*f
  - …
  - $=T(n/2^i) + i * f$
- When i=lg n, $T(n/2^i)=T(1)=e$, and thus,
  - T(n) = e + f * lgn
- Drop low order terms and ignore leading constants
  - T(n) = Θ(lgn)

Mini-quiz (on Moodle)

- Try the repeated substitution method on
- T(n)=
  - a   if n=1
  - 2T(n/2) + b   otherwise, i.e., if n>1

Eq. A.5, CLRS p. 1147

- which is only slightly different from what we have done for
- T(n)=
  - e   if n=1
  - T(n/2) + f   otherwise, i.e., if n>1

- Let's check whether it is still Θ(lgn)? If not, what is the complexity?

- $T(n)= 2T(\frac{n}{2}) + b$
- $=2(2T(\frac{n/2}{2})+b)+b = 2^2T(\frac{n}{2^2})+(2+1)b$
- $=2^2(2T(\frac{n}{2^3}))+b)+(2+1)b =2^3T(\frac{n}{2^3})+(2^2+2+1)b$
- $= 2^3(2T(\frac{n}{2^4})+b)+(2^2+2+1)b = 2^4T(\frac{n}{2^4})+(2^3+2^2+2+1)b$
- $= 2^iT(\frac{n}{2^i})+(2^{i-1}+\ldots+2^3+2^2+2+1)b$
- $= 2^iT(\frac{n}{2^i})+ b * \sum_{k=0}^{i-1} 2^k$
- $= 2^iT(\frac{n}{2^i})+ b \frac{2^{(i-1)+1}-1}{2-1}$
- $= 2^iT(\frac{n}{2^i}) + b (2^i-1)$

Geometric series

For real $x \neq 1$, the summation

$$\sum_{k=0}^{n} x^k = 1 + x + x^2 + \cdots + x^n$$

is a *geometric* or *exponential series* and has the value

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1}-1}{x-1} .$$

CLRS p. 1147

- $T(n) = 2^i T(\frac{n}{2^i}) + b\,(2^i\text{-}1)$
- To use T(1)=a, we set $n/2^i=1$ and get i=lgn.
- $T(n)=2^{\lg n}\, a + b\,(2^{\lg n}\text{-}1)$
- =a*n + b*n – b
- =(a+b)n-b
- $=\Theta(n)$

a = cost of solving trivial case, b = cost of dividing.

Merge Sort

- An algorithm that is able to solve the sorting problem and uses the divide-and-conquer technique.
- Assume that we are going to sort a sequence of numbers in array A.
- **Divide**
  - If A has at least two elements (nothing needs to be done if A has zero or one elements), remove all the elements from A and put them into two sequences, A_1 and A_2 , each containing about half of the elements of A. (i.e. A1 contains the first ┌n/2┐ elements and A2 contains the remaining └ n/2┘ elements).
- **Conquer**
  - Sort sequences A1 and A2 using Merge Sort.
- **Combine**
  - Put back the elements into A by merging the sorted sequences A1 and A2 into one sorted sequence

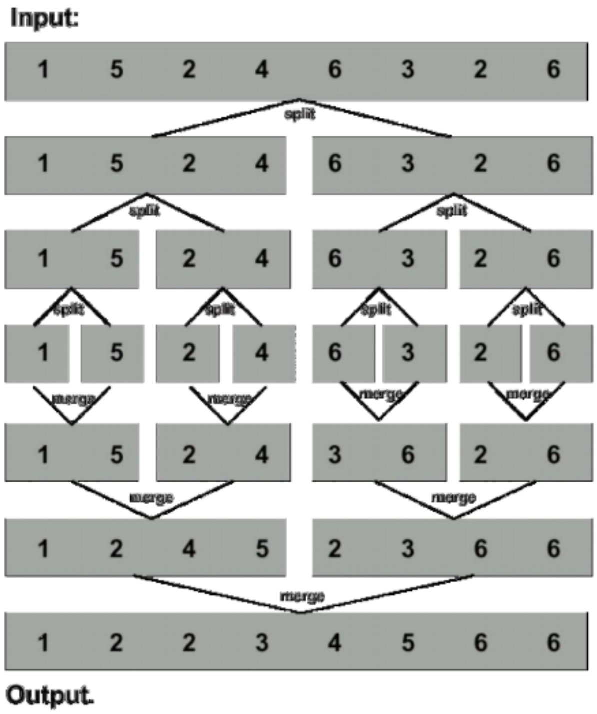
Merge Sort: Algorithm

```
Merge-Sort(A, p, r)
    if p < r then
        q=⌊(p+r)/2⌋
        Merge-Sort(A, p, q)
        Merge-Sort(A, q+1, r)
        Merge(A, p, q, r)
```

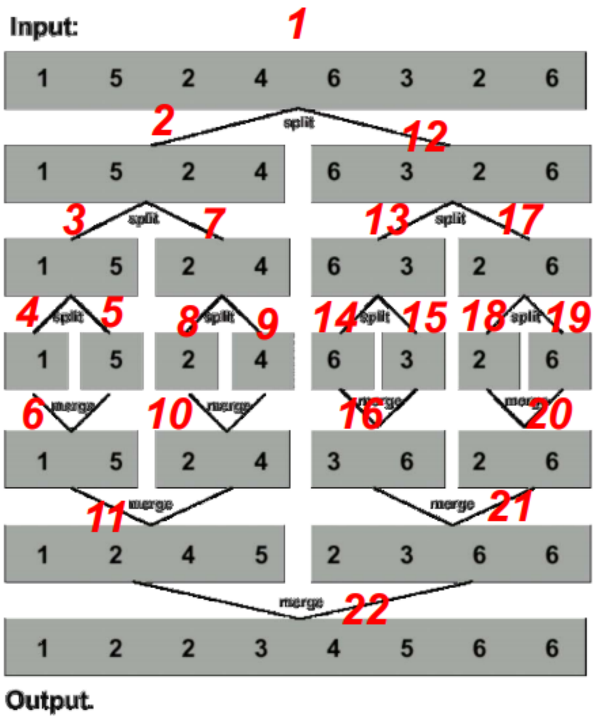
```
Merge(A, p, q, r)
    Take the smallest of the two topmost elements of
sequences A[p..q] and A[q+1..r] and put into the
resulting sequence. Repeat this, until both sequences
are empty. Copy the resulting sequence into A[p..r].
```

Merge Sort Summarized

- To sort *n* numbers
  - if n=1, done!
  - recursively sort 2 lists of numbers ⌈n/2⌉ and ⌊n/2⌋ elements
  - merge 2 sorted lists in $\Theta(n)$ time
- Strategy
  - break problem into similar (smaller) subproblems
  - recursively solve subproblems
  - combine solutions to answer



1. Merge-Sort(A, 1, 8)
2. Merge-Sort(A, 1, 4)
3. Merge-Sort(A, 1, 2)
4. Merge-Sort(A, 1, 1)
5. Merge-Sort(A, 2, 2)
6. Merge(A, 1, 1, 2)
7. Merge-Sort(A, 3, 4)
8. Merge-Sort(A, 3, 3)
9. Merge-Sort(A, 4, 4)
10. Merge(A, 3, 3, 4)
11. Merge(A, 1, 2, 4)
12. Merge-Sort(A, 5, 8)
13. Merge-Sort(A, 5, 6)
14. Merge-Sort(A, 5, 5)
15. Merge-Sort(A, 6, 6)
16. Merge(A, 5, 5, 6)
17. Merge-Sort(A, 7, 8)
18. Merge-Sort(A, 7, 7)
19. Merge-Sort(A, 8, 8)
20. Merge(A, 7, 7, 8)
21. Merge(A, 5, 6, 8)
22. Merge(A, 1, 4, 8)



51

Running Time of Merge Sort

- Write the recurrences
  - Solving the trivial problem: constant time, Θ(1)
  - Dividing: constant time, Θ(1)
  - Combining: linear time, Θ(n)
  - Each division, we get two sub-problems with half size.
- Thus, we have T(n)=
  - Θ(1) if n=1
  - $2T(n/2) + \Theta(n)$ if n>1

Mini Quiz

- What is the running time of merge sort?

- By solving the recurrence
- T(n)=
  - $\Theta(1)$ if n=1
  - $2T(n/2) + \Theta(n)$ if n>1

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

$$\begin{aligned}
T(n) &= 2T(n/2) + n && \text{substitute} \\
&= 2\big(2T(n/4) + n/2\big) + n && \text{expand} \\
&= 2^2 T(n/4) + 2n && \text{substitute} \\
&= 2^2(2T(n/8) + n/4) + 2n && \text{expand} \\
&= 2^3 T(n/8) + 3n && \text{observe the pattern} \\
T(n) &= 2^i T(n/2^i) + in \\
&= 2^{\lg n} T(n/n) + n \lg n = n + n \lg n
\end{aligned}$$

Θ(nlgn)
Better than insertion sort Θ(n^2)

- Can we say merge sort is better than insertion sort?
- Yes, run time Θ(nlgn) vs. Θ(n2)
- No, additional space Θ(n) vs. Θ(1)
    - The merge step requires an additional array with size n
    -
- We will see another sorting algorithm, Heap Sort, that has run time Θ(nlgn) and also Θ(1) additional space.