CSC236 Week 6

Larry Zhang


KEEP
CALM
AND
NEVER
GIVE UP

- Drop date: Nov 9th
    - jk, wait until the test marks are back
- Tutorial this week: algorithm analysis
- PS5 will be out by the end of this week

What we have learned so far

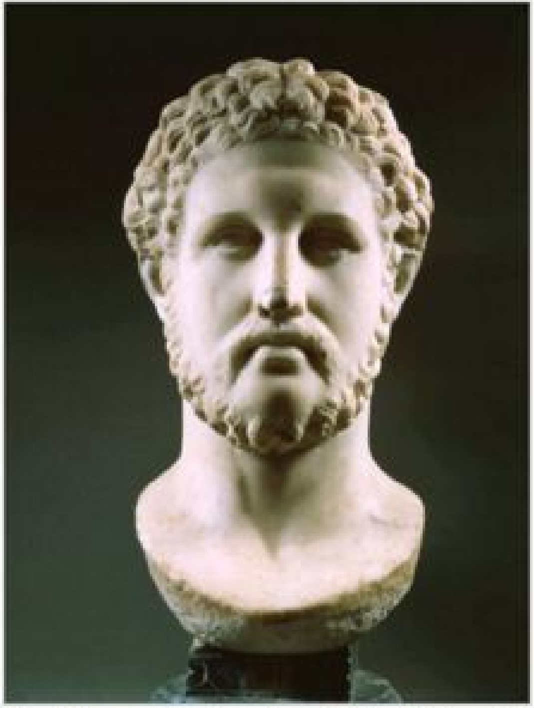
- We learned how to **analyse the runtime** of a recursive algorithm, formally and rigorously.
  - Give a piece of recursive code, **develop the recursive function** that describes its runtime
  - Given the recursive function, find its closed form, using **repeated substitution**.
    - Step 5 of repeated substitution: prove the closed form using **induction**.

With the power of mathematical runtime analysis,
we now have the ability to

design efficient recursive algorithms

Divide-and-Conquer

a common form of recursive algorithm design



Tradition attributes the origin of the motto to Philip of Macedonia: διαίρει καὶ βασίλευε *diaírei kài basíleue*, in ancient Greek: «divide and rule»

Divide and Conquer: Overall structure

- **Divide**: divide the problem into two or more smaller instances of the same problem (subproblems)
- **Conquer**: if the subproblem is small enough, return the solution directly; otherwise, solve it recursively.
- **Combine**: combine the solutions to the subproblems to solve the original problem.

Learn by example

Maximum Segment Sum

- Given a list, e.g., **[2, -5, 8, -6, 10, -2]**
- A segment is a contiguous portion of the list
- The maximum segment sum is the maximum sum of any segment.
- What's the maximum segment sum of the list above?
  - **It is 12.** [2, -5, <u>8, -6, 10</u>, -2]
- **We will solve this problem using divide-and-conquer**

Now let’s divide-and-conquer this problem

- **Divide**: divide the problem into two or more smaller instances of the same problem (subproblems)
- **Conquer**: if the subproblem is small enough, return the solution directly; otherwise, solve it recursively.
- **Combine**: combine the solutions to the subproblems to solve the original problem.
- **Try it yourself first!**

[2, -5, 8, -6, 10, -2]

A[low..high]

Divide

- Divide the problem into two or more smaller instances of the same problem (subproblems)

[2, -5, 8, -6, 10, -2]

[2, -5, 8] **[-6, 10, -2]**

A[low..high]

mid = (low+high) // 2

A[low..mid] **A[mid+1..high]**

Conquer

- if the subproblem is small enough, return the solution directly; otherwise, solve it recursively.
- When is subproblem **small enough**?
  - when the list has only one element, i.e., when **low==high**, e.g., **A[3..3]**
  - directly return **max(A[low], 0)**
  - **# empty segments gives 0, which is better than negative**
- When it's not small enough, solve the subproblems recursively
  - call **max_seg_sum(A, low, mid)**, and
  - call **max_seg_sum(A, mid+1, high)**

Combine

- **Combine**: combine the solutions to the subproblems to solve the original problem.
- sol_left = max_seg_sum(A, low, mid)
- sol_right = max_seg_sum(A, mid+1, high)
- **So what is the solution to the original problem?**
- **max(sol_left, sol_right). Right?**
- **NO!**

What's missing?

[2, -5, <u>8, -6, 10</u>, -2]

[2, -5, <u>8]</u> **<u>[-6, 10</u>, -2]**

The solution to the original problem is neither the solution to the **left subproblem** nor the solution to the **right subproblem**.

The real solution from the segment that **crosses the middle point**!

The correct solution should be the maximum of the three:

- solution to the left subproblem: max_seg_sum(A, low, mid)
- solution to the right subproblem: max_seg_sum(A, mid+1, high)
- solution that crosses the middle point: **max_crossing(A, low, mid, high)**

max_crossing(A[low..high])

[2, -5, 8]   [-6, 10, -2]

- Start from **mid**, extend the segment all the way to the **left**, get the maximum segment sum for the left side
- **PLUS**
- Start from **mid**, extend the segment all the way to the **right**, get the maximum segment sum for the right side

```
def max_crossing(A, low, mid, high):
    left_sum = 0
    s = 0
    for i in range(mid, low - 1, -1):
        s = s + A[i]
        if s > left_sum:
            left_sum = s
    right_sum = 0
    s = 0
    for i in range(mid + 1, high + 1):
        s = s + a[i]
        if s > right_sum:
            right_sum = s
    return left_sum + right_sum
```

What is the worst-case runtime of max_crossing()?

- goes from **mid** all the way to **low** end of A, and
- goes from **mid** all the way to **high** end of A
- Basically, traverse the whole length of A, namely **n**.
- For each entry of A, do some constant work **d**.
- So overall it is, ***dn***.

The correct solution should be the maximum of the three:

- solution to the left subproblem: **max_seg_sum(A, low, mid)**
- solution to the right subproblem: **max_seg_sum(A, mid+1, high)**
- solution that crosses the middle point: **max_crossing(A, low, mid, high)**

max_seg_sum, the complete algorithm

```
def max_seg_sum(A, low, high):
    if low == high:
        return max(A[low], 0)
    mid = (low + high) // 2
    left_sum = max_seg_sum(A, low, mid)
    right_sum = max_seg_sum(A, mid + 1, high)
    cross_sum = max_crossing(A, low, mid, high)
    return max(left_sum, right_sum, cross_sum)
```

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn + e & n > 1 \end{cases}$$

Runtime analysis:

- Let n be the length of A
- if n = 1
  - constant **c**
- if n > 1
  - get mid takes constant time ***e***
  - ***two*** recursive calls of max_seg_sum
  - each with input size ***n/2***
  - so ***2T(n/2)***
  - max_crossing take ***dn***, as discussed before

Just do the math

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn + e & n > 1 \end{cases}$$

$$T(n) = dn\log_2 n + (c+e)n - e$$

$$\in \Theta(n \log n)$$

Design algorithms like a pro

- Know the philosophy, e.g., divide and conquer
- Develop algorithm based on the philosophy
- Analyse the runtime of the algorithm, know how fast it is before even trying it.
- Change anything in the algorithm, know its exact impact to the runtime.

Example 2: MergeSort

MergeSort

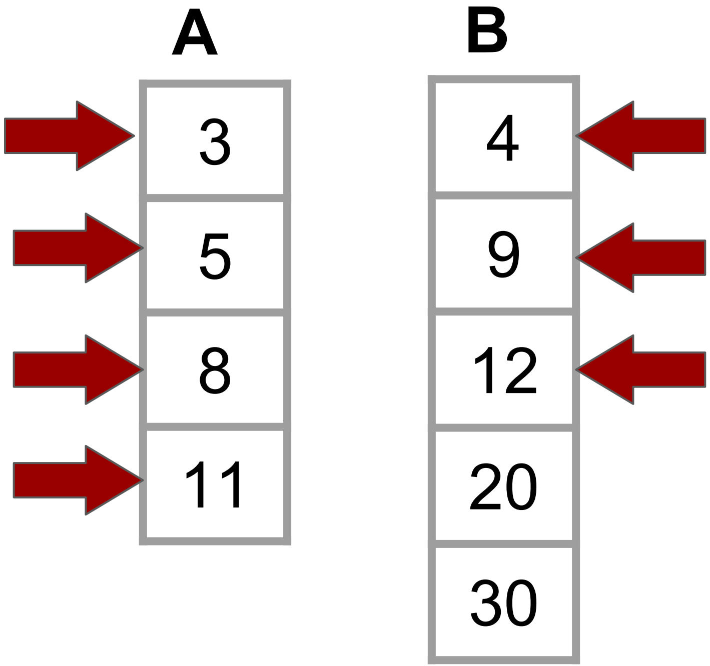
Another typical divide-and-conquer algorithm

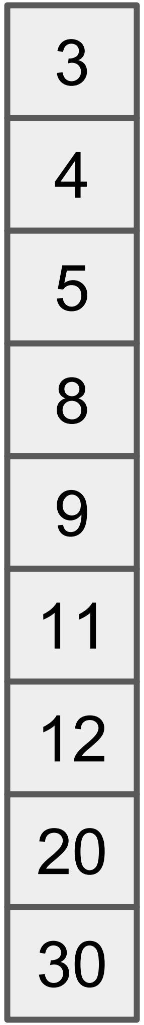
- Divide: divide the list into two equal halves
- Conquer: recursively sort the two halves
- Combine: merge the two sorted halves into a sorted whole

```
def mergesort(A):
    if len(A) == 1:
        return A
    else:
        m = len(A) // 2
        L1 = mergesort(A[0..m-1])
        L2 = mergesort(A[m..len(A)-1])
    return merge(L1, L2)
```

How does this merge work?
How much time does it take?

Merging two sorted lists





The procedure

- compare the two pointed by arrows
- add the smaller one to the output
- advance the arrow of the smaller one
- When reaching the end of one list, append everything left in the other

Worst-case runtime:

c(len(A) + len(B))

or, cn

```
def merge(A, B):
    i = 0    # the arrow for A
    j = 0    # the arrow for B
    C = []   # the output list
    while i < len(A) and j < len(B):
        if A[i] <= B[j]:
            C.append(A[i])
            i += 1
        else:
            C.append(B[j])
            j += 1
    return C + A[i..len(A)-1] + B[j..len(B)-1]
```

```
def mergesort(A):
    if len(A) == 1:
        return A
    else:
        m = len(A) // 2
        L1 = mergesort(A[0..m-1])
        L2 = mergesort(A[m..len(A)-1])
    return merge(L1, L2)
```

Runtime of MergeSort (recursive function):

$$T(n) = \begin{cases} c & n = 1 \\ 2T(n/2) + dn + e & n > 1 \end{cases}$$

Exactly the same as max_seg_sum!

Do the same math again ...

...substitute…

...guess…

...solve…

...plug…

...prove...

MergeSort's worst-case runtime is in Θ(n log n)

Takeaway

- From ***2T(n/2) + dn + e***
- To ***Θ(nlogn)***
- Now it takes the whole process of repeated substitution
    - i.e., 5~6 steps
- There is a quicker way, which takes just 1 step.
- We will learned it next week.