COMP122/20 - Data Structures and Algorithms

20 Divide-and-Conquer, Mergesort and Quicksort

Instructor : Ke Wei 〖柯韋〗

➽ A319 ✆ Ext. 6452 ✉ wke@ipm.edu.mo

http://brouwer.ipm.edu.mo/COMP122/20/

Bachelor of Science in Computing, School of Applied Sciences, Macao Polytechnic Institute

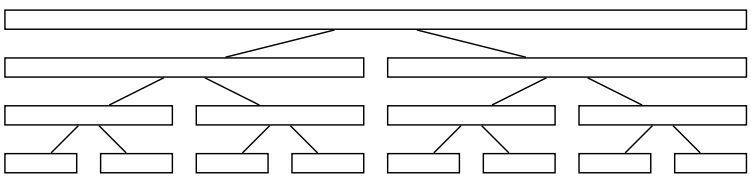
April 13, 2020

Outline

👁 *Textbook §12.2 – 12.3.*

Divide-and-Conquer

- A problem is divided into smaller problems and solved recursively.
- The conquering phase consists of patching together the solutions.
- If we can divide a problem into equal sizes, then the sub-problems can be reduced to base cases in logarithmic time.

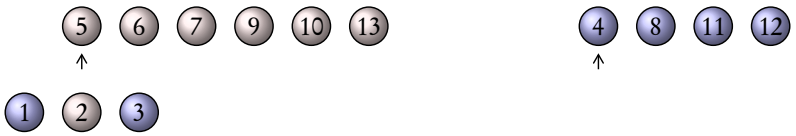


- For sorting problems, divide-and-conquer means to divide the list into smaller parts of approximately equal sizes and sort them recursively, then combine the sorted sub-lists into one sorted list.

Mergesort

The fundamental operation in the Mergesort algorithm is merging two sorted lists. Because the lists are sorted, they can be scanned in one pass.

- Merging two arrays cannot be done by swapping elements, therefore in-place merging for arrays is generally *impossible*. A third array is required.
- For linked lists, we can link elements arbitrarily so that in-place merging is possible.

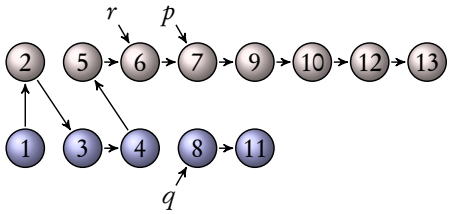


Once we have the merging algorithm, sorting is easy:

- Divide the list into two parts of equal length (left half and right half).
- Recursively sort the two halves.
- Merge the two lists into one sorted list.

Merging Two Sorted Linked Lists

- We don't want to rely on dummy nodes, otherwise we must allocate a temporary dummy node when breaking a list into two. We can use forward links only to merge two linked lists.
- The function *merge* takes two pointers p, q each pointing to the first node of a sorted linked list, and two integers m, n specifying the lengths respectively, and returns a pointer to the merged linked list.



Merging Two Sorted Linked Lists — Code

The *merge* function takes two ordered linked lists p and q, and returns the merged list, the two original lists are destroyed.

```
1  def merge(p, q):
2      if not p: return q
3      elif not q: return p
4      else:
5          if p.elm <= q.elm: s, p = p, p.nxt
6          else: s, q = q, q.nxt
7          r = s # s : first node, r : current last node
8          while p and q:
9              if p.elm <= q.elm: r.nxt, r, p = p, p, p.nxt
10             else: r.nxt, r, q = q, q, q.nxt
11         r.nxt = p if p else q
12         return s
```

Mergesort on Singly Linked Lists

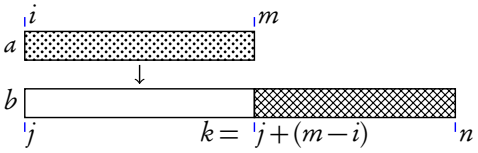
The *mergesort_l* function takes a linked list with head node h, (1) divides the list; (2) recursively sorts each part; (3) finally merges the sorted sub-lists and returns the head node of the merged list.

```
def mergesort_l(h):
    if h and h.nxt:
        r, q = h, h.nxt
        while q and q.nxt:
            r, q = r.nxt, q.nxt.nxt
        q = r.nxt
        r.nxt = None
        return merge(mergesort_l(h), mergesort_l(q))
    else:
        return h
```


Merging Two Arrays

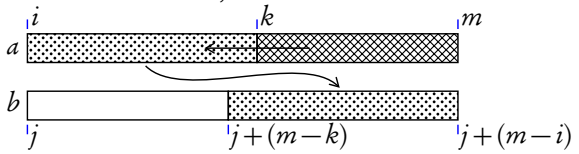
Suppose we have a segment of an array-based list b from j to n, where the front $m-i$ cells are vacant and the rest contains sorted elements, then we can merge another sorted segment of an array-based list a from i to m into the segment of b.

```
def merge_seg(a, i, m, b, j, n):
    k = j+(m-i)
    while i < m and k < n:
        if not b[k] <= a[i]: b[j], j, i = a[i], j+1, i+1
        else: b[j], j, k = b[k], j+1, k+1
    while i < m:
        b[j], j, i = a[i], j+1, i+1
```



Mergesort-Copying an Array-Based List

Suppose we have such a function to transfer an array a's segment to another place in b while sorting, we can recursively use it to fulfill itself, as shown.

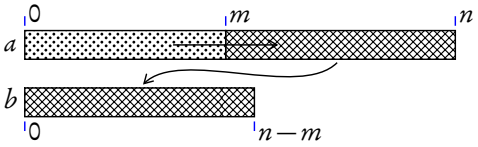



```
def mergesort_copy(a, i, m, b, j):
    if m-i > 1:
        k = (m+i+1)//2 # k starts the smaller rear half
        mergesort_copy(a, i, k, b, j+(m-k))
        mergesort_copy(a, k, m, a, i)
        merge_seg(a, i, i+(m-k), b, j, j+(m-i))
    elif m > i:
        b[j] = a[i]
```

Mergesort on Array-Based Lists

Finally, we can mergesort the first n elements of an array a by (1) mergesort-copying the rear half to auxiliary space b; (2) mergesort-copying the front half to the rear half; (3) merging back the sorted front half from auxiliary space.

```
1 def mergesort(a):
2     n = len(a)
3     if n > 1:
4         m = n//2 # m starts the bigger rear half
5         h = n-m
6         b = [None]*h
7         mergesort_copy(a, m, n, b, 0)
8         mergesort_copy(a, 0, m, a, h)
9         merge_seg(b, 0, h, a, 0, n)
```



Analysis of Mergesort — Space

For an array of n elements, we need $\lceil \frac{n}{2} \rceil$ auxiliary space for mergesort.

- $\mathcal{O}(n)$ auxiliary space for array-mergesort.

For a linked list of n elements, we need non-tail recursive calls of depth $\log n$. However, list mergesort can be done from bottom-up, by repeatedly merging all adjacent pairs of k-sized sub-lists, for $k = 1, 2, 4, 8, \ldots, \frac{n}{2}$, without recursive calls. Try to figure out how to program it.

- $\mathcal{O}(\log n)$ auxiliary space for top-down list-mergesort.
- $\mathcal{O}(1)$ auxiliary space for bottom-up list-mergesort.

Analysis of Mergesort — Time

For an array or a list of n elements, we count the number of element moves (or re-linkings). Let $T(n)$ be the number of moves, we have two sub-mergesorts of $\frac{n}{2}$ elements and one merge of n elements, that

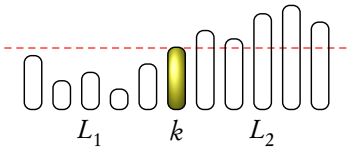
$$T(n) = 2T\left(\frac{n}{2}\right) + n \implies$$
$$\frac{T(n)}{n} = \frac{T\left(\frac{n}{2}\right)}{\frac{n}{2}} + 1 = \frac{T\left(\frac{n}{4}\right)}{\frac{n}{4}} + 1 + 1 = \cdots = \frac{T(1)}{1} + \underbrace{1 + 1 + \cdots + 1}_{\log n} \implies$$
$$T(n) = n + n \log n.$$

- Mergesort takes overall $\mathcal{O}(n \log n)$ time to sort a list of size n.

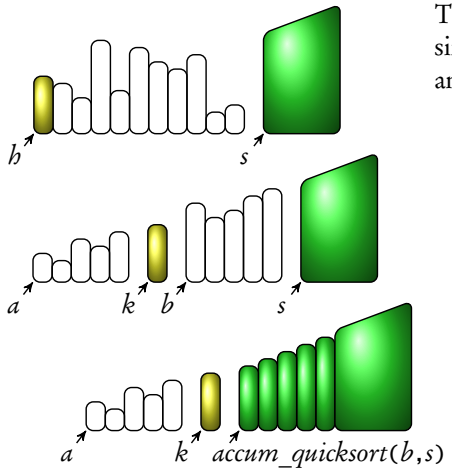
Mergesort is stable, since it's easy to trace which half going to which part of a merge and adjust the code accordingly to select the element from the original front half if there are elements of equal key.

Quicksort

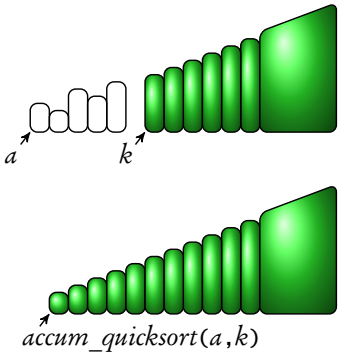
- If we don't need to merge the sorted lists, then the combining phase will be simpler.
- Pick up any element k in a list L as the *pivot*.
- Partition $L - \{k\}$ into two parts L_1 and L_2, where the elements in L_1 are less than (or equal to) k while the elements in L_2 are greater than (or equal to) k.
- Return recursively sorted L_1 followed by k followed by recursively sorted L_2.
- To make L_1 and L_2 of similar sizes, how to choose the pivot is important. Picking up the first element is easy, especially for linked lists, but can lead to the worst case easily.



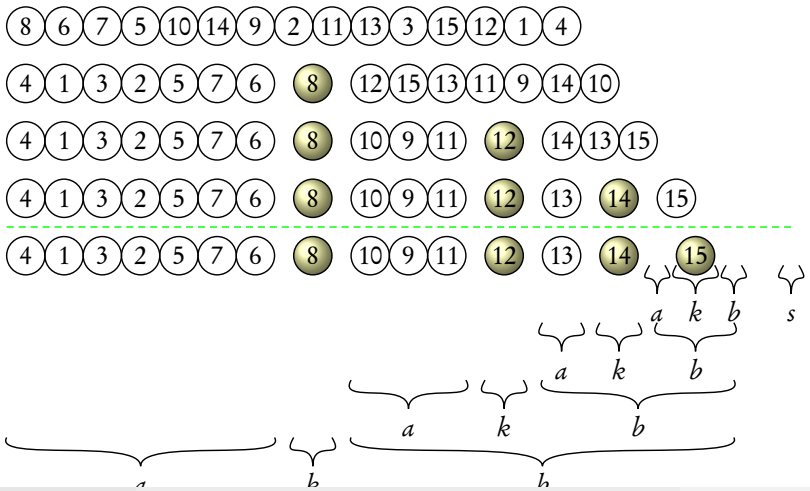
Quicksort on Singly Linked Lists — Illustrated



The *accum_quicksort*(h, s) function sorts a singly linked list h, and insert it in front of another linked list s.



Quicksort on Singly Linked Lists — Example



Quicksort on Singly Linked Lists — Code

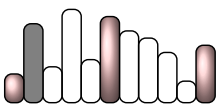
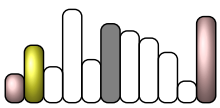
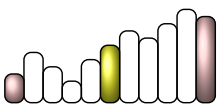
```
def accum_quicksort(h, s):
    if h:
        a = b = None
        k, h = h, h.nxt
        while h:
            p, h = h, h.nxt
            if p.elm <= k.elm:
                p.nxt, a = a, p
            else:
                p.nxt, b = b, p
        k.nxt = accum_quicksort(b, s)
        return accum_quicksort(a, k)
    else:
        return s
```


The Problem of Pivot Choice

- If the input is presorted or in reverse order, then the strategy being used in *accum_quicksort* provides a poor partition, because either all the elements go into L_1 or they go into L_2. This happens consistently throughout the recursive calls.
- This is the worst case of quicksort: quadratic — $\mathcal{O}(n^2)$ — time to do nothing.
- Although presorted lists are unlikely, but smaller presorted sub-lists in sub-problems are highly possible.
- The quicksort on linked lists is much slower than the mergesort.

Choosing the Pivot from Median of 3

- For arrays, we can access the elements randomly at no cost. We may sample the first, the last and the center elements, and pick up the median of the three as the pivot.
- We may even sort these three elements.
- We partition the array by putting the smaller elements to the left side, the larger elements to the right side.
- We store the pivot to one end temporarily, so that it is not staying in the partitioning process. Later, we restore it to the position between the two partitions.

Median of 3 — Code

From array elements $a[s]$ to $a[n-1]$, *median_of_3* rearranges the three elements and returns the selected pivot.

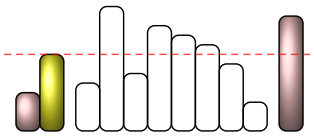
```
1  def median_of_3(a, s, n):
2      c, r = (s+n)//2, n-1
3
4      if a[s] > a[c]:
5          a[s], a[c] = a[c], a[s]
6      if a[s] > a[r]:
7          a[s], a[r] = a[r], a[s]
8      if a[c] > a[r]:
9          a[c], a[r] = a[r], a[c]
10
11     a[s+1], a[c] = a[c], a[s+1]
12     return a[s+1]
```


Quicksort on Array-Based Lists

The *quicksort* function sorts the elements from $a[s]$ to $a[n-1]$.

```
1  def quicksort(a, s, n):
2      if n-s <= 1: return
3
4      k = median_of_3(a, s, n)
5      if n-s <= 3: return  # ...
```

Given a pivot, to partition an array of elements, we scan from both ends towards the middle until the scans collide. During the scans, elements occurring incorrectly are swapped.



Quicksort on Array-Based Lists (2)

```
6      i, j = s+2, n-2
7      while True:
8          while a[i] < k:
9              i += 1
10         while a[j] > k:
11             j -= 1
12         if i < j:
13             a[i], a[j] = a[j], a[i]
14             i, j = i+1, j-1
15         else:
16             break
17     a[s+1], a[j] = a[j], a[s+1]
18     quicksort(a, s, j)
19     quicksort(a, j+1, n)
```

Analysis of Quicksort — Time

We count the number of element comparisons.

- *Worst case* : when in every partitioning, the pivot is of the minimum or the maximum key, one of the partition contains all the elements except the pivot, thus there are $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$ comparisons, i.e. $\mathcal{O}(n^2)$.
- *Average case* : if we choose pivots randomly, each pivot has a chance of $\frac{1}{2}$ to partition off at least $\frac{1}{4}$ elements.

The expected size of a partition at the recursive call of depth d is at most $\left(\frac{3}{4}\right)^{\frac{d}{2}} n$, we have the expected depth of the recursive calls to be $2\log_{\frac{4}{3}} n$. Thus, we have expectedly $\mathcal{O}(n \log n)$ comparisons.

Analysis of Quicksort — Space

For a list of n elements, we need the amount of auxiliary space proportional to the depth of the recursive calls.

- *Worst case* : $\mathcal{O}(n)$ auxiliary space.
- *Average case* : $\mathcal{O}(\log n)$ auxiliary space.

Quicksort on linked lists can be made stable, if we change LIFO to FIFO when building partitions. Quicksort on arrays is *not* stable, since a latter element can be swapped to the front in the scans.

Summary of Sorting Algorithms

Name	Worst Time	Average Time	Auxiliary Space on Arrays	In Place on Arrays	Linked Lists	Stable on Arrays
Insertion Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Yes	Yes
Selection Sort	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(1)$	Yes	Yes	No
Heapsort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(1)$	Yes	No	No
Mergesort	$\mathcal{O}(n \log n)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(n)$	No	Yes	Yes
Quicksort	$\mathcal{O}(n^2)$	$\mathcal{O}(n \log n)$	$\mathcal{O}(\log n)$	Yes	Yes	No