comp10002
Foundations of Algorithms

Semester Two, 2025

Problem Solving, Algorithms, and More

Lecture slides prepared by Alistair Moffat

Overview

Problem solving strategies

Abstract data types

Hashing

Mergesort

Priority queues

Heapsort

Chapter 9 – Concepts

A range of generic approaches to solving problems:

- Generate and test
- Divide and conquer
- Simulation
- Approximation
- Adaptation

Generate and test

If solution space can be enumerated and mapped onto a sequence of integers, then systematically try candidate answers until one that meets specified criteria is found.

If solution space is infinite, then include a loop counter and "no answer found" exit point.

If solution space is multiply-infinite, be sure that the dimensions are treated fairly.

- `isprimefunc.c`

Divide and conquer

- Break the problem into smaller instance(s)
- Solve the instance(s), perhaps recursively
- Combine solutions to create solution to original problem.

For example, the towers of Hanoi problem yields to a recursive divide-and-conquer approach:

- `hanoi.c`
- `quicksort.c`
- plus another sorting method, coming real soon

Divide and conquer – Subset sums

Is there a subset of the following numbers that adds up to exactly 1,000?

```
 34,  38,  39,  43,  55,  66,  67,  84,  85,  91,
101, 117, 128, 138, 165, 168, 169, 182, 184, 186,
234, 238, 241, 276, 279, 288, 386, 387, 388, 389,
413, 444, 487, 513, 534, 535, 616, 722, 786, 787
```

In general, given n integer values, and a target value k, determine if there is a subset of the integers that adds up to exactly k.

Divide and conquer – Subset sums

Evaluate all subsets of the `n` items, and if any one of them adds to `k`, return "yes".

Either the last value `A[n-1]` is part of the sum, or it is not.

If it is, a subset sum on the first `n-1` items in the array must add to `k-A[n-1]`.

- `subsetsum.c`

Hard problems – An aside

The subset sum problem is an example of a very important class of related hard problems.

All (so far!) take exponential time in size of input, which makes them intractable. We can solve for small problems, and we can quickly check any proposed solution for large problems. But we can't find solutions for large problems.

If any one of these problems can be solved in polynomial time, then they all can!

Monte Carlo methods

Use pseudo-random number generation to allow modeling of a physical system.

Function `srand()` is used to initialize the random-number sequence; then each call to `rand()` returns the next seemingly-unrelated `int` in the sequence.

- `gamble1.c`
- `gamble2.c`
- `random.c`
- `montecarlo.c`
- `drunk.c`

Approximation

Solve a simpler problem, with a known degree of fidelity relative to the original one, or with a clear strategy of estimating the extent of the error.

Many numerical examples: integration, curve fitting, root finding, solutions to DEs and PDEs.

Need to be very careful with the floating point arithmetic being performed. *Numerical Analysis* is the branch of mathematics/computing that studies such problems.

- `linelength.c`
- `bisection.c`
- `spring.c`

Adaptation

Modify the solution or approach already used for another similar problem.

Principled "borrowing".

Always give a full attribution, and recognize the previous work honestly.

Chapter 9 – Summary

All algorithms start somewhere, with someone saying "surely there is a better way than that".

Complexity theory is the branch of computing which seeks to prove, for a given problem, that "no there can't be".

Chapter 12 (Part II) – Concepts

Abstract data types: algorithms tend to require related sets of operations providing packages of functionality. There is usually choice in how to implement them.

The queue and stack abstract data types are one example. There are several different implementation options.

The priority queue is another category of data structure, also with multiple implementation options.

The dictionary abstract data type is a third. We have already seen a range of implementation options, and have another important one coming soon.

Queue abstract data type

Operations to be supported:

- $Q \leftarrow queue\_create\_empty()$
- $queue\_is\_empty(Q)$
- $Q' \leftarrow queue\_append(Q, item)$
- $item \leftarrow queue\_head(Q)$
- $Q' \leftarrow queue\_tail(Q)$

Implementation options: circular array, linked list.

All operations should take $O(1)$ time.

Stack abstract data type

Operations to be supported:

- $S \leftarrow stack\_create\_empty()$
- $stack\_is\_empty(S)$
- $S' \leftarrow stack\_push(S, item)$
- $item \leftarrow stack\_top(S)$
- $S' \leftarrow stack\_pop(S)$

Implementation options: array, linked list.

All operations should take $O(1)$ time.

Priority queue abstract data type

Operations to be supported:

- $Q \leftarrow pq\_create\_empty()$
- $pq\_is\_empty(Q)$
- $Q' \leftarrow pq\_insert(Q, item, priority)$
- $item \leftarrow pq\_max\_priority(Q)$
- $Q' \leftarrow pq\_delete\_max(Q)$

Implementation options: ???

Operation cost: ????

Dictionary abstract data type

Operations to be supported:

- $D \leftarrow dict\_create\_empty()$
- $dict\_is\_empty(D)$
- $D' \leftarrow dict\_insert(D, item, key)$
- $item \leftarrow dict\_search(D, key)$
- $D' \leftarrow dict\_delete\_item(D, key)$

plus (maybe):

- $item \leftarrow dict\_smallest(D)$
- $item' \leftarrow dict\_next\_element(D, item)$

Dictionary data structures

Structure	*Insert*	*Search*	*Delete*	*Smallest*	*Next*
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(1)$

Times are worst case unless indicated. *Delete* and *Next* follow a *Search*, *Smallest*, or *Next* operation that establishes a "current" item.

Dictionary data structures

Structure	*Insert*	*Search*	*Delete*	*Smallest*	*Next*
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(1)$
Unsorted linked list	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$

Times are worst case unless indicated. *Delete* and *Next* follow a *Search*, *Smallest*, or *Next* operation that establishes a "current" item. Note the error in Table 12.2 in the textbook.

Dictionary data structures

Structure	*Insert*	*Search*	*Delete*	*Smallest*	*Next*
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(1)$
Unsorted linked list	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$	???	$O(n)$	$O(n)$

Times are worst case unless indicated. *Delete* and *Next* follow a *Search*, *Smallest*, or *Next* operation that establishes a "current" item. Note the error in Table 12.2 in the textbook.

Dictionary data structures

Structure	*Insert*	*Search*	*Delete*	*Smallest*	*Next*
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(1)$
Unsorted linked list	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$	???	$O(n)$	$O(n)$
– average case	$O(\log n)$	$O(\log n)$	???	$O(\log n)$	$O(\log n)$

Times are worst case unless indicated. *Delete* and *Next* follow a *Search*, *Smallest*, or *Next* operation that establishes a "current" item. Note the error in Table 12.2 in the textbook.

Dictionary data structures

Structure	*Insert*	*Search*	*Delete*	*Smallest*	*Next*
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(1)$
Unsorted linked list	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$	???	$O(n)$	$O(n)$
– average case	$O(\log n)$	$O(\log n)$	???	$O(\log n)$	$O(\log n)$
Balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)^*$

Times are worst case unless indicated. *Delete* and *Next* follow a *Search*, *Smallest*, or *Next* operation that establishes a "current" item. Note the error in Table 12.2 in the textbook. Balanced search trees are not part of comp10002, take comp20007 next year.

Dictionary data structures

Structure	*Insert*	*Search*	*Delete*	*Smallest*	*Next*
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(1)$
Unsorted linked list	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$	???	$O(n)$	$O(n)$
– average case	$O(\log n)$	$O(\log n)$	???	$O(\log n)$	$O(\log n)$
Balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)^*$
Hashing	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$

Times are worst case unless indicated. *Delete* and *Next* follow a *Search*, *Smallest*, or *Next* operation that establishes a "current" item. Note the error in Table 12.2 in the textbook. Balanced search trees are not part of comp10002, take comp20007 next year.

Dictionary data structures

Structure	*Insert*	*Search*	*Delete*	*Smallest*	*Next*
Unsorted array	$O(1)$	$O(n)$	$O(n)$	$O(n)$	$O(n)$
Sorted array	$O(n)$	$O(\log n)$*	$O(n)$	$O(1)$	$O(1)$
Unsorted linked list	$O(1)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
Sorted linked list	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(1)$
Binary search tree	$O(n)$	$O(n)$	???	$O(n)$	$O(n)$
– average case	$O(\log n)$	$O(\log n)$	???	$O(\log n)$	$O(\log n)$
Balanced search tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(1)^*$
Hashing	$O(n)$	$O(n)$	$O(1)$	$O(n)$	$O(n)$
– average case	$O(1)$	$O(1)$	$O(1)$	$O(n)$	$O(n)$

Times are worst case unless indicated. *Delete* and *Next* follow a *Search*, *Smallest*, or *Next* operation that establishes a "current" item. Note the error in Table 12.2 in the textbook. Balanced search trees are not part of comp10002, take comp20007 next year.

Hashing

Idea: use a hash function $h()$ to deterministically construct a seemingly-random integer in $[0 \ldots t-1]$ from each possible key value, for some value t such that $0.5 \lesssim n/t \lesssim 4$.

Use those integers to index an array A of size t, putting x in location $A[h(x)]$.

If $n \leq t$ and very[10] lucky, the set of n keys does not generate any collisions. If so, A can just be an array of items.

But probably won't be lucky – the birthday paradox. To handle collisions, A should be an array of buckets.

Hashing – Collisions

Each bucket is a secondary dictionary structure that supports *insert*, *search*, and so on. For example, can use a linked list (known as hashing with separate chaining) as the secondary structure.

Or the buckets can be realloc'ed arrays that are created and doubled as required.

Assuming random hashing into $0 \ldots (t-1)$, each bucket contains approximately $n/t = O(1)$ items.

As n grows, n/t also increases. At upper limit $n/t \approx 4$, multiply t by eight, `realloc()` array A to match, and rehash all of the n items into the larger set of buckets.

Hash functions – Good and bad

Use all characters of the key and their positions too in some way, before a final mod t operation to generate a value in the range $0 \ldots (t-1)$.

Never just add character values, or shift-n-add, that isn't enough "randomness". And always make the hash function itself depend on an initial seed, so it is reusable.

Then always test the hash function on typical data, checking the distribution of bucket loadings or the comparison count.

- `hashing.c`
- `hashingbad.c`
- `hashscaffold.c`

Divide and conquer sorting

Already saw Quicksort. There is another beautiful divide-and-conquer sorting algorithm:

```
mergesort(A[0...n-1])
    if n ≤ 1
        return
    mid ← (n/2)
    mergesort(A[0...mid-1])
    mergesort(A[mid...n-1])
    merge(A[0...mid-1], A[mid...n-1])
```

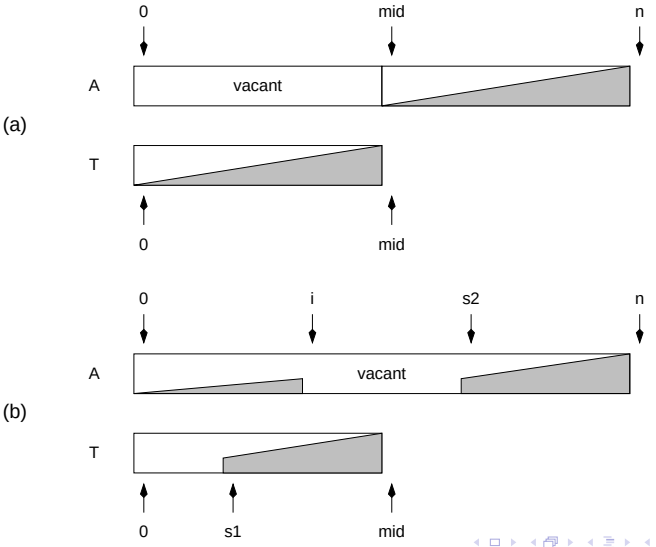
Where Quicksort is "hard split, easy join", this one is "easy split, hard join".

Mergesort – Merging

$merge(A[0 \ldots mid-1], A[mid \ldots n-1])$
    $T[0 \ldots mid-1] \leftarrow A[0 \ldots mid-1]$
    $i, s1, s2 \leftarrow 0, 0, mid$
    while $s1 < mid$ **and** $s2 < n$
        if $T[s1] < A[s2]$
            $A[i] \leftarrow T[s1]$
            $i, s1 \leftarrow i+1, s1+1$
        else
            $A[i] \leftarrow A[s2]$
            $i, s2 \leftarrow i+1, s2+1$
    $A[i \ldots s2-1] \leftarrow T[s1 \ldots mid-1]$

Needs a temporary array T half the size of original input. Same array can be used in all recursive calls.

Mergesort – Merging



Mergesort – Analysis

Counting the comparisons required:

$$T(n) = \begin{cases} 1 & \text{if } n \leq 1 \\ M(n) + T(\lfloor n/2 \rfloor) + T(\lceil n/2 \rceil) & \text{if } n > 1 \end{cases}$$

where $M(n)$ is the cost of merging.

With $M(n) = n - 1$, $T(n) = n \log_2 n - n + 1$ for n a power of two.

That is, Mergesort takes $O(n \log n)$ time in the worst case.

Brilliant! ... except for the extra space. Can we have it all?

Heaps

In an array $A[0 \ldots n-1]$, define the two *children* of item $A[i]$ to be in $A[2i+1]$ and $A[2i+2]$.

With the *parent* of $A[i]$ in $A[(i-1)/2]$.

This balanced tree is implicit, based on array positions. The root is in $A[0]$, and the leaves in $A[n/2 \ldots n-1]$. No extra space is required for pointers.

Add one more requirement to make it a heap – no child may be larger than its parent.

Heaps – Example

Initially (with $n = 9$):

13 16 14 10 15 17 18 30 25

Converted into a heap:

30 25 18 16 15 17 14 10 13

How? And how long does it take?

Heaps – Construction

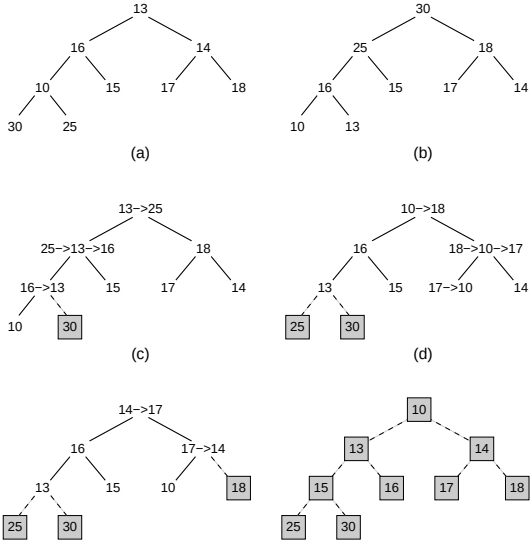

```
build_heap(A[0...n-1])
    for i ← n/2 - 1 downto 0
        sift_down(A, i, n)

sift_down(A, p, n)
    child ← 2p + 1
    if child < n
        if child + 1 < n and A[child] < A[child + 1]
            child ← child + 1
        if A[p] < A[child]
            swap(A[p], A[child])
            sift_down(A, child, n)
```

Heaps – Heapsort!

And now, for the moment of truth ...

$heapsort(A[0 \ldots n-1])$
    $build\_heap(A[0 \ldots n-1])$
    for $a \leftarrow n-1$ **downto** 1
        $swap(A[0], A[a])$
        $sift\_down(A, 0, a)$

Heapsort – Example



Heapsort – Analysis

Phase 1 (*build_heap*):

$$T_1(n) = 0\frac{n}{2} + 2\frac{n}{4} + 4\frac{n}{8} + 6\frac{n}{16} + \cdots + (2\log n)\frac{n}{n} = 2n$$

Phase 2 (swapping maximums):

$$T_2(n) = n \cdot (2\log n)$$

In-place sorting in $O(n \log n)$ time in the worst case.

Triumph? It doesn't get any better!

Priority queue implementation options

The heap is one implementation option for the priority queue abstract data type.

The *pq_insert*() and *pq_delete_max*() operations take $O(\log n)$ time each when the queue contains n items.

The *pq_max_priority*() operation takes $O(1)$ time.

Sorting algorithms

Attribute	*Insertionsort*	*Quicksort*	*Mergesort*	*Heapsort*
Random input	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
– average case		$O(n \log n)$		

Times are worst case unless indicated.

Sorting algorithms

Attribute	*Insertionsort*	*Quicksort*	*Mergesort*	*Heapsort*
Random input	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
– average case		$O(n \log n)$		
Nearly sorted input	$O(n+?)$	$O(n^2)$	$O(n+?)^*$	$O(n \log n)$
– average case		$O(n \log n)$		

Times are worst case unless indicated.

Sorting algorithms

Attribute	*Insertionsort*	*Quicksort*	*Mergesort*	*Heapsort*
Random input	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
– average case		$O(n \log n)$		
Nearly sorted input	$O(n+?)$	$O(n^2)$	$O(n+?)^*$	$O(n \log n)$
– average case		$O(n \log n)$		
Additional space	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$

Times are worst case unless indicated.

Sorting algorithms

Attribute	*Insertionsort*	*Quicksort*	*Mergesort*	*Heapsort*
Random input	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
– average case		$O(n \log n)$		
Nearly sorted input	$O(n+?)$	$O(n^2)$	$O(n+?)^*$	$O(n \log n)$
– average case		$O(n \log n)$		
Additional space	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$
Implementation	simple	needs care	needs care	moderate

Times are worst case unless indicated.

Sorting algorithms

Attribute	*Insertionsort*	*Quicksort*	*Mergesort*	*Heapsort*
Random input – average case	$O(n^2)$	$O(n^2)$ $O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Nearly sorted input – average case	$O(n+?)$	$O(n^2)$ $O(n \log n)$	$O(n+?)^*$	$O(n \log n)$
Additional space	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$
Implementation	simple	needs care	needs care	moderate
When $n = 10^3$	fast	fast	fast	fast

Times are worst case unless indicated.

Sorting algorithms

Attribute	*Insertionsort*	*Quicksort*	*Mergesort*	*Heapsort*
Random input	$O(n^2)$	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
– average case		$O(n \log n)$		
Nearly sorted input	$O(n+?)$	$O(n^2)$	$O(n+?)^*$	$O(n \log n)$
– average case		$O(n \log n)$		
Additional space	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$
Implementation	simple	needs care	needs care	moderate
When $n = 10^3$	fast	fast	fast	fast
When $n = 10^7$	dreadful	fast	fastest	fast

Times are worst case unless indicated.

Sorting algorithms

Attribute	*Insertionsort*	*Quicksort*	*Mergesort*	*Heapsort*
Random input – average case	$O(n^2)$	$O(n^2)$ $O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Nearly sorted input – average case	$O(n+?)$	$O(n^2)$ $O(n \log n)$	$O(n+?)^*$	$O(n \log n)$
Additional space	$O(1)$	$O(\log n)$	$O(n)$	$O(1)$
Implementation	simple	needs care	needs care	moderate
When $n = 10^3$	fast	fast	fast	fast
When $n = 10^7$	dreadful	fast	fastest	fast
Linked lists?	only doubly	singly	singly	no

Times are worst case unless indicated.

Chapter 12 (Part II) – Program examples

- `heapsort.c`
- `mergesort.c`

Chapter 12 (Part II) – Summary

Algorithms are fun!

And there are plenty more where these ones came from . . .