

comp10002 Foundations of Algorithms

Semester Two, 2019

Problem Solving, Algorithms, and More

© The University of Melbourne, 2019
Lecture slides prepared by Alistair Moffat

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

Problem solving strategies

More dictionaries

Sorting again

Priority queues

A range of generic approaches to solving problems:

- ▶ Generate and test *explorer all possibility until find a solution*
- ▶ Divide and conquer
- ▶ Simulation
- ▶ Approximation
- ▶ Adaptation

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`

- ▶ 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

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 .

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

use recursion

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`

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!**

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`

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

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`

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

Principled “borrowing”.

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

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”.

Abstract data types: different algorithms require different suites of operations.

The **dictionary** abstract data type is an important one,

The **priority queue** is another.

And as a final triumph for the book, and for the subject:
 $O(n \log n)$ worst-case time sorting.

Operations to be supported:

- ▶ $D \leftarrow \text{make_empty}()$
- ▶ $D' \leftarrow \text{insert}(D, \text{key}, \text{item})$
- ▶ $\text{item_ptr} \leftarrow \text{search}(D, \text{key})$
- ▶ $D' \leftarrow \text{delete}(D, \text{key})$

plus (maybe):

- ▶ $\text{item_ptr} \leftarrow \text{find_smallest_key}(D)$
- ▶ $\text{item_ptr} \leftarrow \text{find_next_key}(D)$

Array? But search and ordered processing are slow.

Sorted array? But insert is slow.

Binary search tree? But not robust, and hence vulnerable.

Balanced search tree? Yes, can do $O(\log n)$ time for all operations, but complex to implement. (And you have to take another subject.)

Hashing: insert, search, and delete in $O(1)$ average time. No ordered processing, but still useful for some applications.

①
unsorted array
search $O(n)$
insert $O(1)$
delete $O(n)$
smallest $O(n)$
next-smallest $O(n)$

②

Hashing

comp10002
Foundations of
Algorithms

lec11

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

search $O(\log n) \rightarrow$ binary search
insert $O(n) \rightarrow$ you have to find its place

delete $O(n)$

smallest $O(1)$

next-smallest $O(1)$.

③ Unsorted linked list

search $O(n)$

insert $O(1)$

delete $O(1)$

smallest $O(n)$

next-smallest $O(n)$

Main idea: take each key, and use a **hash function** $h()$ to deterministically construct a seemingly-random integer in a constrained range $[0 \dots t - 1]$, where $t > n$, the number of items to be stored.

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

If lucky, every thing fits just fine, and no **collisions**.

Don't expect to be lucky – the birthday “paradox”.

next smallest

Hashing

comp10002
Foundations of
Algorithms

lec11

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

④ sorted linked list

search $O(L)$

insert

search $+O(1)$

delete

search $+O(1)$

smallest

$O(1)$

next smallest $O(1)$

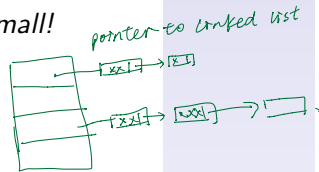
If the location indicated by the hash function is used, put the item "somewhere else" that makes sense.

When searching for x , look first in the location $A[h(x)]$ indicated by the hash function.

If not there, look in all possible "somewhere else" locations.

If not there, then not in the dictionary.

Try to keep the set of "somewhere else" options small!



⑤ binary search tree

$O(\log n)$ avg

$O(n)$ worst

Hashing – Collisions

comp10002
Foundations of
Algorithms

lec11

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

Search
insert
delete
smallest
next smallest

search $O(1)$
???
 $O(\log n)$ avg
[$O(1)$ avg] ?

① Hashing
search insert, delete $O(1)$ * avg * hashing operation
smallest next smallest $O(n)$.

When the location indicated by $h(x)$ is already occupied:

- ▶ **linear advance**: move cyclicly forward to the next vacant cell in the array (simple, but not that good, chains get longer, and deletion is problematic)
- ▶ **cuckoo hashing**: displace the previous object to allow the new one to be in its right spot, then re-insert it using a different hash function (effective, but needs a suite of hash functions, still needs complex deletion)
- ▶ **separate chaining**: use a secondary data structure to maintain the set of colliding objects: linked list, tree, or dynamic array (may require additional overhead space).

② balanced

search $O(\log n)$
insert $O(\log n)$
 $O(\log n)$
 $O(\log n)$
 $O(1)$.

Separate chaining is easy to implement, and robust against overflow – doesn't require that $h > n$ the way the other two methods do.

When records are large, overhead of extra pointers per item is small.

Whenever table **loading** gets too high, double the size of the array of lists, rehash all current objects, and resume.

Use all characters of the key, and seek to involve all bit positions in the intermediate value, before a final `%t` operation to bring into the right range.

Never just add character values, or shift-n-add. It isn't enough "randomness", and performance will degrade.

Test the hash function on the data it will be applied to, by checking the distribution of bucket loadings.

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

If hash function is good (and if data is not pathological), after n objects are inserted, average list length is n/t items long. (And can do stats on standard deviation and etc.)

If $n/t < K$ for some K , then insert and search are $O(1)$ – provided that the hash function itself can be evaluated in $O(1)$ time per key. Might not be possible for strings.

Further enhancement – move any item that is accessed to the front of its list. Accelerates search, unless access pattern is uniform.

Divide and conquer sorting

comp10002
Foundations of
Algorithms

lec11

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

Already saw [Quicksort](#). There is another beautiful divide-and-conquer sorting algorithm:

```
mergesort( $A[0 \dots n - 1]$ )  
  if  $n \leq 1$   
    return  
   $mid \leftarrow (n/2)$   
  mergesort( $A[0 \dots mid - 1]$ )  
  mergesort( $A[mid \dots n - 1]$ )  
  merge( $A[0 \dots mid - 1], A[mid \dots n - 1]$ )
```

Where Quicksort is “hard split, easy join”, this one is “easy split, hard join”.

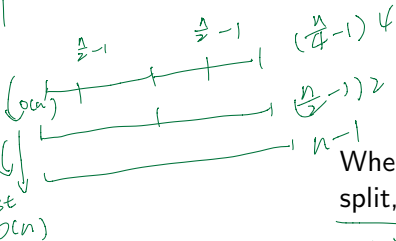
one comparison \rightarrow one join

$O(n \log n)$

length $O(n)$

depth $\log n$

length



actually $n-1$ comparison

```
merge( $A[0 \dots mid - 1]$ ,  $A[mid \dots n - 1]$ )  
   $T[0 \dots mid - 1] \leftarrow A[0 \dots mid - 1]$  → copy into new array  
   $i, s1, s2 \leftarrow 0, 0, mid$   
  while  $s1 < mid$  and  $s2 < n$   
    if  $T[s1] < A[s2]$   
       $A[i] \leftarrow T[s1]$   
       $i, s1 \leftarrow i + 1, s1 + 1$   
    else  
       $A[i] \leftarrow A[s2]$   
       $i, s2 \leftarrow i + 1, s2 + 1$   
   $A[i \dots s2 - 1] \leftarrow T[s1 \dots mid - 1]$ 
```

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

Mergesort – Merging

comp10002
Foundations of
Algorithms

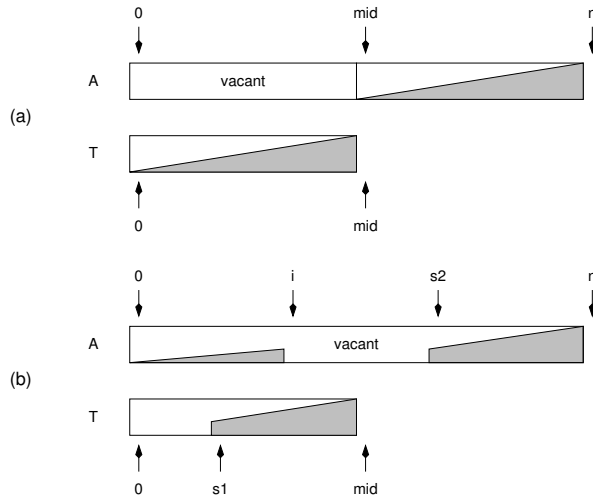
lec11

Problem solving
strategies

Dictionaries

Sorting again

Priority queues



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. *one comparison → throw one thing.*

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

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

left child $2i+1$
right child $2i+2$

In an array $A[0 \dots n-1]$, 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 $A[n/2 \dots n-1]$. No extra space is required for pointers.

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

↑
biggest on the heap
make sure it's a heap

Heaps – Example

comp10002
Foundations of
Algorithms

lec11

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

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?

```
build_heap( $A[0 \dots n - 1]$ )  
  for  $i \leftarrow n/2 - 1$  downto 0  
    sift_down( $A, i, n$ )
```

→ make every node
satisfies the properties
of heap

```
sift_down( $A, p, n$ )  
  child  $\leftarrow 2p + 1$   
  if child  $< n$   
    if child + 1  $< n$  and  $A[\text{child}] < A[\text{child} + 1]$   
      child  $\leftarrow \text{child} + 1$   
    if  $A[p] < A[\text{child}]$   
      swap( $A[p], A[\text{child}]$ )  
      sift_down( $A, \text{child}, n$ )
```

• construct a heap

• Delete the min element n times.

And now, for the moment of truth ...

```
heapsort( $A[0 \dots n - 1]$ )  
  build_heap( $A[0 \dots n - 1]$ )  
  for  $a \leftarrow n - 1$  downto 1  
    swap( $A[0]$ ,  $A[a]$ )  
    sift_down( $A$ , 0,  $a$ )
```

Heapsort – Example

comp10002
Foundations of
Algorithms

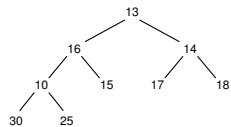
lec11

Problem solving
strategies

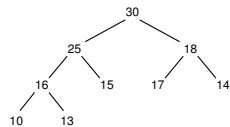
Dictionaries

Sorting again

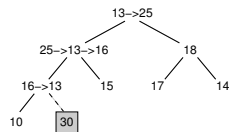
Priority queues



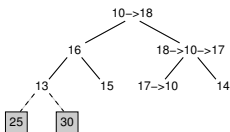
(a)



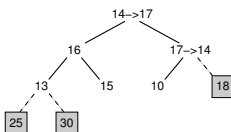
(b)



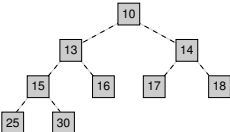
(c)



(d)



(e)



(f)

Phase 1 (*build_heap*): *1 level below, we do 2 comparisons*

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

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!

Chapter 12 (Part II) – Program examples

comp10002
Foundations of
Algorithms

lec11

Problem solving
strategies

Dictionaries

Sorting again

Priority queues

▶ `heapsort.c`

▶ `mergesort.c`

Algorithms are fun!

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