Advanced Algorithmics (6EAP)

MTAT.03.238

Linear structures, sorting, searching, etc

Jaak Vilo
2015 Fall

Big-Oh notation classes

<table>
<thead>
<tr>
<th>Class</th>
<th>Informal</th>
<th>Intuition</th>
<th>Analogy</th>
</tr>
</thead>
<tbody>
<tr>
<td>f(n) ∈ o(g(n))</td>
<td>f is dominated by g</td>
<td>Strictly below</td>
<td>&lt;</td>
</tr>
<tr>
<td>f(n) ∈ O(g(n))</td>
<td>Bounded from above</td>
<td>Upper bound</td>
<td>≤</td>
</tr>
<tr>
<td>f(n) ∈ Θ(g(n))</td>
<td>“equal to”</td>
<td>Bounded from above and below</td>
<td>=</td>
</tr>
<tr>
<td>f(n) ∈ Ω(g(n))</td>
<td>f dominates g</td>
<td>Strictly above</td>
<td>&gt;</td>
</tr>
</tbody>
</table>

Conclusions

• Algorithm complexity deals with the behavior in the long-term
  — worst case
  — average case
  — best case

• In practice, long-term sometimes not necessary
  — E.g. for sorting 20 elements, you don’t need fancy algorithms...

Physical ordered list ~ array

• Memory /address/
  — Garbage collection

• Files (character/byte list/lines in text file,...)

• Disk
  — Disk fragmentation

Linear, sequential, ordered, list ...

Memory, disk, tape etc – is an ordered sequentially addressed media.

Linear data structures: Arrays

• Array
• Bidirectional map
• Bit array
• Bit field
• Bitboard
• Bitmap
• Circular buffer
• Control table
• Image
• Dynamic array
• Gap buffer
• Hashed array tree
• Heightmap
• Lookup table
• Matrix
• Parallel array
• Sorted array
• Sparse array
• Sparse matrix
• Iliffe vector
• Variable-length array
Linear data structures: Lists

- Doubly linked list
- Array list
- Linked list
- Self-organizing list
- Skip list
- Unrolled linked list
- VList

Lists:

- Xor linked list
- Zipper
- Doubly connected edge list
- Difference list

Lists:

- Array

```c
L = int[MAX_SIZE]
L[2] = 7
L[size++] = new
L[3] = 7
L[++size] = new
```

Multiple lists, 2-D-arrays, etc...

2D array

```
&AI[j] = A + i*(nr_el_in_row*el_size) + j*el_size
```

Linear Lists

- Operations which one may want to perform on a linear list of \( n \) elements include:
  - gain access to the \( k \)th element of the list to examine and/or change the contents
  - insert a new element before or after the \( k \)th element
  - delete the \( k \)th element of the list

Abstract Data Type (ADT)

- High-level definition of data types
- An ADT specifies
  - A collection of data
  - A set of operations on the data or subsets of the data
- ADT does not specify how the operations should be implemented
- Examples
  - vector, list, stack, deque, priority queue, table (map), associative array, set, graph, digraph

ADT

- A datatype is a set of values and an associated set of operations
- A datatype is abstract if it is completely described by its set of operations regardless of its implementation
- This means that it is possible to change the implementation of the datatype without changing its use
- The datatype is thus described by a set of procedures
- These operations are the only thing that a user of the abstraction can assume

Abstract data types:

- Dictionary
- Stack (LIFO)
- Queue (FIFO)
- Queue (double-ended)
- Priority queue (fetch highest-priority object)
- ... (key,value)

Dictionary

- Container of key-element (k,e) pairs
- Required operations:
  - insert( k,e ),
  - remove( k ),
  - find( k ),
  - isEmpty()
- May also support (when an order is provided):
  - closestKeyBefore( k ),
  - closestElemAfter( k )
- Note: No duplicate keys

Some data structures for Dictionary ADT

- Unordered
  - Array
    - Sequence/ist
- Ordered
  - Array
    - Sequence (Skip Lists)
    - Binary Search Tree (BST)
    - AVL trees, red-black trees
    - (2, 4) Trees
    - B-Trees
- Valued
  - Hash Tables
  - Extendible Hashing
### Primitive & composite types

#### Primitive types
- Boolean (for boolean values: True/False)
- Char (for character values)
- Int (for integral or fixed-precision values)
- Float (for storing real number values)
- Double (a larger size of type float)
- String (for string of chars)
- Enumerated type

#### Composite types
- Array
- Record (also called tuple or struct)
- Union
- Tagged union (also called a variant, variant record, discriminated union, or disjoint union)
- Plain old data structure

### Linear data structures

#### Arrays
- Gap buffer
- Hashed array tree
- Heightmap
- Lookup table
- Matrix
- Parallel array
- Sorted array
- Sparse array
- Sparse matrix
- Riffle vector
- Variable-length array

#### Lists
- Doubly linked list
- Linked list
- Self-organizing list
- Skip list
- Unrolled linked list
- VList
- Xor linked list
- Zipper
- Doubly connected edge list

### Trees...

- Binary tree
- Binary search tree
- AVL tree
- AA tree
- Randomized Cartesian tree
- Heaps
- AF-heap
- Fibonacci heap
- Binomial heap
- Binary heap
- Bx Queap
- 2-3 tree
- Dancing tree
- B*-tree
- B-tree
- Weight-balanced Treap
- Trie
- Tries
- D-ternary heap
- 2-3 heap
- 3-4 heap
- 4-5 heap
- Ternary heap
- Trie
- Fenwick tree
- Exponential tree
- Enfilade
- Fusion tree
- Disjoint-set data structure
- Spaghetti stack
- Link/cut tree
- Y-fast
- X-fast
- 2-3 heap
- Trees
- Finger tree
- Exponential minimax tree
- Alternating decision tree
- Syntax tree
- Rapidly-exploring random tree
- BSP tree
- Bounding box
- BK tree
- VP tree
- M tree
- Trees
- Decision tree
- Hashes,
- Hashes,
- Graphs,
- Other
- Graphs
- Adjacency list
- Adjacency matrix
- Graph-structured stack
- Scene graph
- Binary decision diagram
- Zero suppressed decision diagram
- And-inverter graph
- Directed graph
- Directed acyclic graph
- Propositional directed acyclic graph
- Multigraph
- Hypergraph

### Hashes, Graphs, Other

- Hashes
- Bloom filter
- Distributed hash table
- Hash array mapped trie
- Hash list
- Hash table
- Hash tree
- Hash trie
- Koode
- Prefix hash tree
- Hashes
- Graphs
- Adjacency list
- Adjacency matrix
- Graph-structured stack
- Scene graph
- Binary decision diagram
- Zero suppressed decision diagram
- And-inverter graph
- Directed graph
- Directed acyclic graph
- Propositional directed acyclic graph
- Multigraph
- Hypergraph

### Lists: Array

- Access i
- Insert to end
- Delete from end
- Insert
- Delete
- Search

#### Lists: Array

- Access i
- Insert to end
- Delete from end
- Insert
- Delete
- Search

#### Lists: Array

- Access i
- Insert to end
- Delete from end
- Insert
- Delete
- Search

#### Lists: Array

- Access i
- Insert to end
- Delete from end
- Insert
- Delete
- Search
Linear Lists

- Other operations on a linear list may include:
  - determine the number of elements
  - search the list
  - sort a list
  - combine two or more linear lists
  - split a linear list into two or more lists
  - make a copy of a list

Stack

- push(x)  -- add to end (add to top)
- pop()  -- fetch from end (top)

- O(1) in all reasonable cases 😊

- LIFO – Last In, First Out

Linked lists

- head
- tail

Singly linked

Doubly linked

Linked lists: add

- head
- tail

Linked lists: delete

(+) garbage collection?

- head
- tail

Operations

- Array indexed from 0 to n – 1:

<table>
<thead>
<tr>
<th>access/change the kth element</th>
<th>k = 1</th>
<th>1 &lt; k &lt; n</th>
<th>k = n</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert before or after the kth element</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>delete the kth element</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

- Singly-linked list with head and tail pointers

<table>
<thead>
<tr>
<th>access/change the kth element</th>
<th>k = 1</th>
<th>1 &lt; k &lt; n</th>
<th>k = n</th>
</tr>
</thead>
<tbody>
<tr>
<td>insert before or after the kth element</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
<tr>
<td>delete the kth element</td>
<td>O(1)</td>
<td>O(1)</td>
<td>O(1)</td>
</tr>
</tbody>
</table>

under the assumption we have a pointer to the kth node, O(1) otherwise
Improving Run-Time Efficiency

- We can improve the run-time efficiency of a linked list by using a doubly-linked list:

Singly-linked list:

Doubly-linked list:

- Improvements at operations requiring access to the previous node
- Increases memory requirements...

Array indexed from 0 to n - 1:

<table>
<thead>
<tr>
<th></th>
<th>k = 1</th>
<th>1 &lt; k &lt; n</th>
<th>k = n</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert before or after the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>delete the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Singly-linked list with head and tail pointers:

<table>
<thead>
<tr>
<th></th>
<th>k = 1</th>
<th>1 &lt; k &lt; n</th>
<th>k = n</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert before or after the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>delete the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>

Doubly linked list:

Introduction to linked lists: inserting a node

- node *p;
- p = new node;
- p->num = 5;
- p->word = "Ali";
- p->next = NULL

Introduction to linked lists: adding a new node

- How can you add another node that is pointed by p->next?
  - node *q;
  - p = new node;
  - p->num = 5;
  - p->word = "Ali";
  - p->next = NULL;
  - node *q;
  - p->next = q;
  - q->next = NULL

Improving Efficiency

- Consider the following struct definition

```c
struct node {
    string word;
    int num;
    node *next; //pointer for the next node
};
```

node *p = new node;

<table>
<thead>
<tr>
<th></th>
<th>k = 1</th>
<th>1 &lt; k &lt; n</th>
<th>k = n</th>
</tr>
</thead>
<tbody>
<tr>
<td>access/change the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>insert before or after the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>delete the kth element</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
<td>$O(1)$</td>
</tr>
</tbody>
</table>
Introduction to linked lists

```c
node *p, *q;
p = new node;
p->num = 5;
p->word = "Ali";
p->next = NULL;

q = new node;
q->num = 8;
q->word = "Veli";
p->next = q;
q->next = NULL;
```

Pointers in C/C++

```c
p = new node; delete p;
p = new node[20];

p = malloc( sizeof( node ) ); free p;

p = malloc( sizeof( node ) * 20 );
(p+10)->next = NULL; /* 11th elements */
```

Book-keeping

- `malloc, new` – “remember” what has been created, `free(p), delete` (C/C++)
- When you need many small areas to be allocated, reserve a big chunk (array) and maintain your own set of free objects
- Elements of an array of objects can be pointed by the pointer to an object.

Object

- `Object = new object_type`;
- Equals to creating a new object with necessary size of allocated memory (delete can free it)
Some links

- **Pointer basics:** [http://cslibrary.stanford.edu/106/](http://cslibrary.stanford.edu/106/)
- **C++ Memory Management:** What is the difference between malloc/free and new/delete?

Alternative: **arrays and integers**

- If you want to test pointers and linked list etc. data structures, but do not have pointers familiar (yet)
- Use arrays and indexes to array elements instead...

---

Replacing pointers with array index

Maintaining list of free objects

Multiple lists, single free list

Hack: allocate more arrays ...

- use integer division and mod
- AA[ (i-1)/7 ] = AA( i ] % 7 ]
- \( LIST(10) = AA[ 1 ][ 2 ] \)
- \( LIST(19) = AA[ 2 ][ 5 ] \)
Queue

- enqueue(x) - add to end
- dequeue() - fetch from beginning

FIFO – First In First Out

O(1) in all reasonable cases 😊

Circular buffer

- A circular buffer or ring buffer is a data structure that uses a single, fixed-size buffer as if it were connected end-to-end. This structure lends itself easily to buffering data streams.
Stack

- push(x) -- add to end (add to top)
- pop() -- fetch from end (top)
- O(1) in all reasonable cases 😊
- LIFO – Last In, First Out

Stack based languages

- Implement a postfix calculator
  - Reverse Polish notation
- 5 4 3 * 2 - + => 5+((4*3)-2)
- Very simple to parse and interpret
- FORTH, Postscript are stack-based languages

Array based stack

- How to know how big a stack shall ever be?

```
3 6 7 5
3 6 7 5 -
```
- When full, allocate bigger table dynamically, and copy all previous values there
- O(n) ?

- When full, create 2x bigger table, copy previous n elements:

```
3 6 7 5
3 6 7 5 -
```
- After every $2^k$ insertions, perform O(n) copy
- O(n) individual insertions +
- $n/2 + n/4 + n/8 \ldots$ copy-ing
- Total: O(n) effort!

What about deletions?

- when n=32 -> 33 (copy 32, insert 1)
- delete: 33->32
  - should you delete immediately?
  - Delete only when becomes less than 1/4th full
  - Have to delete at least n/2 to decrease
  - Have to add at least n to increase size
  - Most operations, O(1) effort
  - But few operations take O(n) to copy
  - For any m operations, O(m) time

Amortized analysis

- Analyze the time complexity over the entire “lifespan” of the algorithm
- Some operations that cost more will be "covered" by many other operations taking less
Lists and dictionary ADT...

• How to maintain a dictionary using (linked) lists?
• Is k in D?
  – go through all elements d of D, test if d==k  O(n)
  – If sorted: d= first(D); while( d< k ) d=next(D);
  – on average \( n/2 \) tests...
• Add(k,D) => insert(k,D) = O(1) or O(n) – test for uniqueness

Array based sorted list

• is d in D?
• Binary search in D

Binary search – recursive

```java
BinarySearch(A[0..N-1], value, low, high)
{
  if (high < low)
    return -1 // not found
  mid = low + ((high - low) / 2)  // Note: not (low + high) / 2 !
  if (A[mid] > value)
    return BinarySearch(A, value, low, mid-1)
  else if (A[mid] < value)
    return BinarySearch(A, value, mid+1, high)
  else
    return mid // found
}
```

Binary search – iterative

```java
BinarySearch(A[0..N-1], value)
{
  low = 0;  high = N - 1;
  while (low <= high) {
    mid = low + ((high - low) / 2)  // Note: not (low + high) / 2 !
    if (A[mid] > value)
      high = mid - 1
    else if (A[mid] < value)
      low = mid + 1
    else
      return mid // found
  }
  return -1 // not found
}
```

Work performed

• \( x \leftrightarrow A[18] \)? <
• \( x \leftrightarrow A[9] \)? >
• \( x \leftrightarrow A[13] \)? ==

• \( O(\log n) \)

Sorting

• given a list, arrange values so that
  \( L[1] \leq L[2] \leq \ldots \leq L[n] \)
• n elements => n! possible orderings
• One test \( L[i] \leq L[j] \) can divide n! to 2
  – Make a binary tree and calculate the depth
• \( \log(n!) = \Omega(n \log n) \)
• Hence, lower bound for sorting is \( \Omega(n \log n) \)
  – using comparisons...
**Decision tree model**

1. Divide the problem (instance) into subproblems.
2. Conquer the subproblems by solving them recursively.
3. Combine subproblem solutions.

**Proof:** \( \log(n!) = \Omega(n \log n) \)

- \( \log(n!) = \log(n) + \log(n-1) + \log(n-2) + \ldots + \log(1) \)
- \( \geq n/2 \log(n/2) \)
- \( = \Omega(n \log n) \)

**Theorem.** Any decision tree that can sort \( n \) elements must have height \( \Omega(n \log n) \).

**Proof.** The tree must contain \( \geq n! \) leaves, since there are \( n! \) possible permutations. A height-\( h \) binary tree has \( \leq 2^h \) leaves. Thus, \( n! \leq 2^h \).

\[ h \geq \log(n!) \]

\[ \geq \log((n/e)^n) \]

\[ = n \log n - n \log e \]

\[ = \Omega(n \log n) \]
Merge sort

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

It was invented by John von Neumann in 1945.

Example

• Applying the merge sort algorithm:

Merge of two lists: Θ(n)

A, B – lists to be merged
L = new list; // empty
while( A not empty and B not empty )
  if A.first() <= B.first() then append( L, A.first() ); A = rest(A) ;
  else append( L, B.first() ); B = rest(B) ;
append( L, A);  // all remaining elements of A
append( L, B );  // all remaining elements of B
return L

Wikipedia / viz.

Run-time Analysis of Merge Sort

• Thus, the time required to sort an array of size \( n \) > 1 is:
  – the time required to sort the first half,
  – the time required to sort the second half, and
  – the time required to merge the two lists
• That is:
\[
T(n) = \begin{cases} 
\Theta(1) & n = 1 \\
2T(\frac{n}{2}) + \Theta(n) & n > 1 
\end{cases}
\]
Merge sort

- Worst case, average case, best case ...
  \( \Theta(n \log n) \)
- **Common wisdom:**
  - Requires additional space for merging (in case of arrays)
- Homework*: develop in-place merge of two lists implemented in arrays /compare speed/

Quicksort

- Divide-and-conquer algorithm.
- Sorts “in place” (like insertion sort, but not like merge sort).
- Very practical (with tuning).

Divide and conquer

Quick sort an \( n \)-element array:

1. **Divide:** Partition the array into two subarrays around a pivot \( x \) such that elements in lower subarray \( \leq x \) elements in upper subarray \( \geq x \).

2. **Conquer:** Recursively sort the two subarrays.

3. **Combine:** Trivial.

**Key:** Linear-time partitioning subroutine.

Pseudocode for quicksort

```plaintext
QUICKSORT(A, p, r)

if p < r
    then q ← PARTITION(A, p, r)
    QUICKSORT(A, p, q−1)
    QUICKSORT(A, q+1, r)

Initial call: QUICKSORT(A, 1, n)
```

Partitioning subroutine

```plaintext
PARTITION(A, p, q) → A[p...q]  pivot = A[p]

x ← A[p]  i ← p
for j ← p + 1 to q  do if A[j] ≤ x  then i ← i + 1
return i

Invariant: \( x \leq A[i] \leq x \) \( \geq x \)
```

Partitioning version 2

```plaintext
pivot = A[R];
i=L; j=R−1;
while( i<eq j )
    while( A[i] < pivot ) i++ ; // will stop at pivot latest
    while( i<eq j and A[j] >= pivot ) j--; 
    if( i < j ) { swap(A[i],A[j]); i++; j--; }
A[R]=A[i];
A[i]=pivot;
return i;
```
Wikipedia / “video”

Worst-case of quicksort
- Input sorted or reverse sorted.
- Partition around min or max element.
- One side of partition always has no elements.
  \[ T(n) = T(0) + T(n-1) + \Theta(n) \]
  \[ = \Theta(1) + T(n-1) + \Theta(n) \]
  \[ = T(n-1) + \Theta(n) \]
  \[ = \Theta(n^2) \text{ (arithmetic series)} \]

Best-case analysis
(For intuition only!)
If we’re lucky, PARTITION splits the array evenly:
\[ T(n) = 2T(n/2) + \Theta(n) \]
\[ = \Theta(n \log n) \text{ (same as merge sort)} \]

What if the split is always \( \frac{1}{10} \cdot \frac{9}{10} \)?
\[ T(n) = T(\frac{n}{10}) + T(\frac{9n}{10}) + \Theta(n) \]
What is the solution to this recurrence?

Analysis of “almost-best” case
\[ T(\frac{n}{10}) \quad T(\frac{9n}{10}) \]

Analysis of “almost-best” case
\[ \log_{10} n \quad \frac{1}{10} n \quad \frac{9}{10} n \]
\[ \Theta(1) \quad \Theta(1) \]
\[ cn, \log_{10} n \]
\[ (n \log n) \text{ leaves} \]
\[ \Theta(n) \text{ leaves} \]
\[ cn \log_{10} n \leq T(n) \leq cn \log_{10} n + \Theta(n) \]

More intuition
Suppose we alternate lucky, unlucky, lucky, unlucky, lucky, ...
\[ L(n) = 2U(n/2) + \Theta(n) \text{   lucky} \]
\[ U(n) = L(n-1) + \Theta(n) \text{   unlucky} \]

Solving:
\[ L(n) = 2(L(n/2 - 1) + \Theta(n/2)) + \Theta(n) \]
\[ = 2L(n/2 - 1) + \Theta(n) \]
\[ = \Theta(n \log n) \text{   Lucky!} \]

How can we make sure we are usually lucky?
Choice of pivot in Quicksort

- Select median of three ...

- Select random – opponent can not choose the winning strategy against you!

Randomized quicksort

**IDEA:** Partition around a random element.
- Running time is independent of the input order.
- No assumptions need to be made about the input distribution.
- No specific input elicits the worst-case behavior.
- The worst case is determined only by the output of a random-number generator.

Random pivot

Select pivot \( \square \) randomly from the region (blue) and swap with last position
Select pivot as a median of 3 [or more] random values from region
Apply non-recursive sort for array less than 10-20

Quicksort in practice

- Quicksort is a great general-purpose sorting algorithm.
- Quicksort is typically over twice as fast as merge sort.
- Quicksort can benefit substantially from code tuning.
- Quicksort behaves well even with caching and virtual memory.

Randomized quicksort analysis

Let \( T(n) \) = the random variable for the running time of randomized quicksort on an input of size \( n \), assuming random numbers are independent.

For \( k = 0, 1, \ldots, n-1 \), define the **indicator random variable**

\[
X_k = \begin{cases} 
1 & \text{if } \text{PARTITION} \text{ generates a } k : n-k-1 \text{ split,} \\
0 & \text{otherwise.} 
\end{cases}
\]

\( E[X_k] = \Pr(X_k = 1) = \frac{1}{n} \), since all splits are equally likely, assuming elements are distinct.
Analysis (continued)

\[ T(n) = \begin{cases} 
T(0) + T(n-1) + \Theta(n) & \text{if } 0 : n-1 \text{ split}, \\
T(1) + T(n-2) + \Theta(n) & \text{if } 1 : n-2 \text{ split}, \\
\vdots \\
T(n-1) + T(0) + \Theta(n) & \text{if } n-1 : 0 \text{ split}, \\
= \sum_{k=0}^{n-1} X_k (T(k) + T(n-k-1) + \Theta(n)) 
\end{cases} \]

Calculating expectation

\[ E[T(n)] = \sum_{k=0}^{n-1} E[X_k (T(k) + T(n-k-1) + \Theta(n))] \]

Linearity of expectation.

Calculating expectation

\[ E[T(n)] = \sum_{k=0}^{n-1} E[X_k] E[T(k) + T(n-k-1) + \Theta(n)] \]

Independence of \( X_k \) from other random choices.

Calculating expectation

\[ E[T(n)] = \frac{1}{n} \sum_{k=0}^{n-1} E[X_k] (T(k) + T(n-k-1) + \Theta(n)) \]

Linearity of expectation; \( E[X_k] = 1/n \).

Calculating expectation

\[ E[T(n)] = \frac{1}{n} \sum_{k=0}^{n-1} E[X_k] (T(k) + T(n-k-1) + \Theta(n)) \]

\[ = \frac{1}{n} \sum_{k=0}^{n-1} E[T(k)] + \frac{1}{n} \sum_{k=0}^{n-1} E[T(n-k-1)] + \sum_{k=0}^{n-1} \Theta(n) \]

Summations have identical terms.
### Hairy recurrence

\[ E[T(n)] = 2 \sum_{k=2}^{n-1} E[T(k)] + \Theta(n) \]

(The \( k = 0, 1 \) terms can be absorbed in the \( \Theta(n) \).)

**Prove:** \( E[T(n)] \leq an \log n \) for constant \( a > 0 \).

- Choose \( a \) large enough so that \( an \log n \) dominates \( E[T(n)] \) for sufficiently small \( n \geq 2 \).

**Use fact:** \( \sum_{k=2}^{n-1} \log k \leq \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \) (exercise).

---

### Substitution method

\[ E[T(n)] \leq 2 \sum_{k=2}^{n-1} ak \log k + \Theta(n) \]

\[ \leq a \left( \frac{1}{2} n^2 \log n - \frac{1}{8} n^2 \right) + \Theta(n) \]

\[ = an \log n - \left( \frac{an}{4} - \Theta(n) \right) \]

\[ \leq an \log n , \]

if \( a \) is chosen large enough so that \( an/4 \) dominates the \( \Theta(n) \).

---

### Alternative materials

- QuickSort average case analysis
  - [http://eidee/10z](http://eidee/10z)
  - [https://research.cs.wisc.edu/ftp/comp551/website/Courses/quickpages/av10si05.html](https://research.cs.wisc.edu/ftp/comp551/website/Courses/quickpages/av10si05.html)

- [http://eidee/10y](http://eidee/10y) - MIT Open Courseware - Asymptotic notation, Recurrences, Substituteon Master Method

---

### The master method

The master method applies to recurrences of the form

\[ T(n) = a T(n/b) + f(n) \]

where \( a \geq 1, b > 1 \), and \( f \) is asymptotically positive.

---

### 2-pivot version of Quicksort

- (split in 3 regions!)
Three common cases

Compare \( f(n) \) with \( n^{\log_a b} \):
1. \( f(n) = O(n^{\log_a b - \epsilon}) \) for some constant \( \epsilon > 0 \).
   - \( f(n) \) grows polynomially slower than \( n^{\log_a b} \) (by an \( n^\epsilon \) factor).
   \[\text{Solution: } T(n) = \Theta(n^{\log_a b}) .\]
2. \( f(n) = \Theta(n^{\log_a b} \log^k n) \) for some constant \( k \geq 0 \).
   - \( f(n) \) and \( n^{\log_a b} \) grow at similar rates.
   \[\text{Solution: } T(n) = \Theta(n^{\log_a b} \log^{k+1} n) .\]

Three common cases (cont.)

Compare \( f(n) \) with \( n^{\log_a b} \):
3. \( f(n) = \Omega(n^{\log_a b + \epsilon}) \) for some constant \( \epsilon > 0 \).
   - \( f(n) \) grows polynomially faster than \( n^{\log_a b} \) (by an \( n^\epsilon \) factor),
   \[\text{and } f(n) \text{ satisfies the regularity condition that } \alpha f(n/b) \leq c f(n) \text{ for some constant } c < 1.\]
   \[\text{Solution: } T(n) = \Theta(f(n)) .\]

Examples

Ex. \( T(n) = 4T(n/2) + n \)
\[a = 4, \ b = 2 \Rightarrow n^{\log_2 4} = n^2; \ f(n) = n.\]
\[\text{Case 1: } f(n) = O(n^{2 - \epsilon}) \text{ for } \epsilon = 1.\]
\[\therefore T(n) = \Theta(n^2).\]

Ex. \( T(n) = 4T(n/2) + n^2 \)
\[a = 4, \ b = 2 \Rightarrow n^{\log_2 4} = n^2; \ f(n) = n^2.\]
\[\text{Case 2: } f(n) = \Theta(n^2 \log^k n), \text{ that is, } k = 0.\]
\[\therefore T(n) = \Theta(n^2 \log n).\]

Ex. \( T(n) = 4T(n/2) + n^3 \)
\[a = 4, \ b = 2 \Rightarrow n^{\log_2 4} = n^2; \ f(n) = n^3.\]
\[\text{Case 3: } f(n) = \Omega(n^{3 - \epsilon}) \text{ for } \epsilon = 1\]
\[\text{and } 4(n/2)^3 \leq cn^3 \text{ (reg. cond.) for } c = 1/2.\]
\[\therefore T(n) = \Theta(n^3).\]
**Examples**

**Ex.** \( T(n) = 4T(n/2) + n^3 \)

\[ a = 4, \ b = 2 \implies n^{\log_2 4} = n^2; \ f(n) = n^3. \]

**Case 3:** \( f(n) = \Omega(n^{\varepsilon + c}) \) for \( \varepsilon = 1 \)

and \( 4(n/2)^3 \leq cn \) (reg. cond.) for \( c = 1/2. \)

\[ \therefore T(n) = \Theta(n^3). \]

**Ex.** \( T(n) = 4T(n/2) + n^2/\log n \)

\[ a = 4, \ b = 2 \implies n^{\log_2 4} = n^2; \ f(n) = n^2/\log n. \]

Master method does not apply. In particular, for every constant \( \varepsilon > 0, \) we have \( n^\varepsilon = \omega(\log n). \)

---

**Idea of master theorem**

**Recursion tree:**

\[ f(n) \]

\[ f(n/b) \]

\[ \vdots \]

\[ f(n/b^k) \]

\[ \vdots \]

\[ T(n) \]

\[ a \]

\[ f(n/b) \]

\[ (n/b)^2 \]

\[ \vdots \]

\[ (n/b)^k \]

\[ \vdots \]

\[ T(1) \]

**Case 2:** \( (k = 0) \) The weight is approximately the same on each of the \( \log_2 n \) levels.
We can sort in $O(n \log n)$

- Is that the best we can do?

- Remember: using comparisons $<$, $>$, $<=$, $=>$ we can not do better than $O(n \log n)$

How fast can we sort $n$ integers?

- Sort people by their sex? (F/M, 0/1)
- Sort people by year of birth?

Sorting in linear time

**Counting sort:** No comparisons between elements.

- **Input:** $A[1..n]$, where $A[j] \in \{1, 2, \ldots, k\}$.
- **Output:** $B[1..n]$, sorted.
- **Auxiliary storage:** $C[1..k]$.
Loop 1

for $i \leftarrow 1$ to $k$
    do $C[i] \leftarrow 0$

Loop 2

for $j \leftarrow 1$ to $n$
    do $C[A[j]] \leftarrow C[A[j]] + 1$  \( \Rightarrow C[i] = \{|\text{key} = i\|\}

Loop 3

for $i \leftarrow 2$ to $k$
    do $C[i] \leftarrow C[i] + C[i-1]$  \( \Rightarrow C[i] = \{|\text{key} \leq i\|

Loop 4

for $j \leftarrow n$ downto 1
    do $B[C[A[j]]] \leftarrow A[j]$
    \( C[A[j]] \leftarrow C[A[j]] - 1

Analysis

$\Theta(k)$ \( \left\{\begin{align*}
    &\text{for } i \leftarrow 1 \text{ to } k \\
    &\text{do } C[i] \leftarrow 0
\end{align*}\)

$\Theta(n)$ \( \left\{\begin{align*}
    &\text{for } j \leftarrow 1 \text{ to } n \\
    &\text{do } C[A[j]] \leftarrow C[A[j]] + 1
\end{align*}\)

$\Theta(k)$ \( \left\{\begin{align*}
    &\text{for } i \leftarrow 2 \text{ to } k \\
    &\text{do } C[i] \leftarrow C[i] + C[i-1]
\end{align*}\)

$\Theta(n)$ \( \left\{\begin{align*}
    &\text{for } j \leftarrow n \text{ downto } 1 \\
    &\text{do } B[C[A[j]]] \leftarrow A[j] \\
    &C[A[j]] \leftarrow C[A[j]] - 1
\end{align*}\)

$\Theta(n + k)$

Running time

If $k = O(n)$, then counting sort takes $\Theta(n)$ time.

- But, sorting takes $\Omega(n \log n)$ time!
- Where’s the fallacy?

**Answer:**

- **Comparison sorting** takes $\Omega(n \log n)$ time.
- Counting sort is not a **comparison sort**.
- In fact, not a single comparison between elements occurs!
Radix sort

Radix-Sort(A,d)
1. for i = 1 to d /* least significant to most significant */
2. use a stable sort to sort A on digit i

Stable sorting

Counting sort is a stable sort; it preserves the input order among equal elements.

A: 4 1 3 4 3
B: 1 3 3 4 4

Exercise: What other sorts have this property?

Radix sort

- Origin: Herman Hollerith’s card-sorting machine for the 1890 U.S. Census. (See Appendix C.)
- Digit-by-digit sort.
- Hollerith’s original (bad) idea: sort on most-significant digit first.
- Good idea: Sort on least-significant digit first with auxiliary stable sort.

Operation of radix sort

Correctness of radix sort

Induction on digit position
- Assume that the numbers are sorted by their low-order t−1 digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.

Correctness of radix sort

Induction on digit position
- Assume that the numbers are sorted by their low-order t−1 digits.
- Sort on digit t
  - Two numbers that differ in digit t are correctly sorted.
  - Two numbers equal in digit t are put in the same order as the input ⇒ correct order.
Analysis of radix sort

- Assume counting sort is the auxiliary stable sort.
- Sort \( n \) computer words of \( b \) bits each.
- Each word can be viewed as having \( b/r \) base-\( 2^r \) digits.

**Example:** 32-bit word

| 8 | 8 | 8 | 8 |

\( r = 8 \Rightarrow b/r = 4 \) passes of counting sort on base-\( 2^8 \) digits; or \( r = 16 \Rightarrow b/r = 2 \) passes of counting sort on base-\( 2^{16} \) digits.

*How many passes should we make?*

---

Analysis (continued)

Recall: Counting sort takes \( \Theta(n + k) \) time to sort \( n \) numbers in the range from 0 to \( k - 1 \). If each \( b \)-bit word is broken into \( r \)-bit pieces, each pass of counting sort takes \( \Theta(n + 2^r) \) time. Since there are \( b/r \) passes, we have

\[
T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right).
\]

Choose \( r \) to minimize \( T(n, b) \):

- Increasing \( r \) means fewer passes, but as \( r \gg \lg n \), the time grows exponentially.

---

Choosing \( r \)

\[
T(n, b) = \Theta\left(\frac{b}{r} (n + 2^r)\right)
\]

Minimize \( T(n, b) \) by differentiating and setting to 0. Or, just observe that we don’t want \( 2^r \gg n \), and there’s no harm asymptotically in choosing \( r \) as large as possible subject to this constraint. Choosing \( r = \lg n \) implies

\[
T(n, b) = \Theta(b n/\lg n).
\]

- For numbers in the range from 0 to \( n^d - 1 \), we have \( b = d \lg n \Rightarrow \) radix sort runs in \( \Theta(d n) \) time.

---

Conclusions

In practice, radix sort is fast for large inputs, as well as simple to code and maintain.

**Example (32-bit numbers):**

- At most 3 passes when sorting \( \geq 2000 \) numbers.
- Merge sort and quicksort do at least \( \lceil\lg 2000\rceil = 11 \) passes.

**Downside:** Unlike quicksort, radix sort displays little locality of reference, and thus a well-tuned quicksort fares better on modern processors, which feature steep memory hierarchies.

---

Radix sort using lists (stable)

| bible | bible | zabo | wam | aau | cace | cobe | coba | cera |

Radix sort using lists (stable)

```
1.
```

```
a  bible  zabo  cobe
b  bible  zabo
   cobe
  aau
  cobe
```
**Radix sort using lists (stable)**

1. a
   - b
   - c
   - d
2. b
   - a
   - c
   - d
3. c
   - a
   - b
   - d

**Why not from left to right?**

- **Swap ‘0’ with first ‘1’**
- **Idea 1:** recursively sort first and second half
  - Exercise?

---

**Bitwise sort left to right**

- **Idea 2:**
  - swap elements only if the prefixes match...
  - For all bits from most significant
    - advance when 0
    - when 1 -> look for next 0
      - if prefix matches, swap
      - otherwise keep advancing on 0’s and look for next 1

---

```c
void bitwisesort(SORTTYPE *ARRAY, int size)
{
    int i, j, tmp, nrbits;
    register SORTTYPE mask, curbit, group;
    nrbits = sizeof(SORTTYPE) * 8;
    curbit = 1 << (nrbits-1);
    mask = 0;

    do { /* For each bit */
        new_mask:
        for(i=0; i< size && (ARRAY[i] & curbit) == 0; i++) { /* Advance while bit == 0 */
            group = ARRAY[i] & mask; /* Save current prefix snapshot */
            if(i == size) goto new_mask; /* Reached end of array */
            j = i; /* Remember location of 1 */
            if( (ARRAY[j] & mask) != group) goto new_mask; /* New prefix */
            if( (ARRAY[i] & curbit) == 0 ) { /* bit is 0 -> need to swap with previous location of 1 */
                tmp = ARRAY[i]; ARRAY[i] = ARRAY[j]; ARRAY[j] = tmp; /* swap */
                i++; /* swap and increase to the next possible 1 */
            }
        } goto new_mask;

        new_mask:
        for(j = i; (j < size) && !((ARRAY[j] & curbit) == 0); j++) { /* Scan for next 1 */
            if(ARRAY[j] & curbit) { /* notify under mask in new sorted */
                mask = mask | curbit;
            }
        }
    } while(curbit);
    mask = mask | curbit; /* So mask of the already sorted area */
}

Jaak Vilo, Univ. of Tartu
```

---

**Bitwise from left to right**

0010000
0010010
0010100
0011000
1001010
1001001
1001000
1001100
1111100
1111100
1001000
0100100
0101000
0101001
0101000
0101001
0101000
0101001
1001001

- **Swap ‘0’ with first ‘1’**

---

Jaak Vilo, Univ. of Tartu
Bucket sort

- Assume uniform distribution
- Allocate O(n) buckets
- Assign each value to pre-assigned bucket

http://sortbenchmark.org/

- The sort input records must be 100 bytes in length, with the first 10 bytes being a random key
- Minut sort – max amount sorted in 1 minute
  – 116GB in 58.7 sec (Jim Wyllie, IBM Research)
  – 40-node 80-Itanium cluster, SAN array of 2,520 disks
- 2009, 500 GB Hadoop 1406 nodes x (2 Quadcore Xeons, 8 GB memory, 4 SATA)
  Owen O’Malley and Arun Murthy Yahoo Inc.
  - Performance / Price Sort and PennySort
New: The next deadline for submitting entries is September 1, 2015.
We are deprecating and will no longer accept results for PenneySort and the 10<sup>6</sup>, 10<sup>7</sup> an
superceded by CloudSort. 10<sup>8</sup> and 10<sup>9</sup> record JouleSort are too similar to the 10<sup>7</sup> rec
Other than the aforementioned deprecations, there are no rule changes for 2015.
The 2014 records are listed below in green. Thank you to all the 2014 entrants!

**Sort Benchmark**

- [http://sortbenchmark.org/](http://sortbenchmark.org/)
- Sort Benchmark Home Page
- We have a new benchmark called GraySort, named in memory of the father of the sort benchmarks, Jim Gray. It replaces TeraByteSort which is now retired.
- Unlike 2010, we will not be accepting early entries for the 2011 year. The deadline for submitting entries is April 1, 2011.
  - All hardware must be off the shelf and unmodified.
  - For Daytona cluster sorts, where input sampling is used to determine the output partition boundaries, the input sampling must be done evenly across all input partitions.
- New rules for GraySort:
  - The input file size is now minimum ~100TB or 1T records. Entries with larger input sizes also qualify.
  - The winner will have the fastest SortedRecs/Min.
  - We now provide a new input generator that works in parallel and generates binary data. See below.
  - For the Daytonia category, we have two new requirements. (1) The sort must run continuously (repeatedly) for a minimum 1 hour. (This is a new reliability requirement). (2) The system cannot overwrite the input file.

**Order statistics**

- Minimum – the smallest value
- Maximum – the largest value
- In general i<sup>th</sup> value.
- Find the median of the values in the array
- Median in sorted array A:
  - n is odd – \(A[(n+1)/2]\)
  - n is even – \(A[(n+1)/2] \) or \(A[(n+1)/2] \)

**Q: Find i<sup>th</sup> value in unsorted data**

A. O(n)
B. O(n log log n)
C. O(n log n)
D. O(n log<sup>2</sup> n)
Minimum

Minimum(A)
1 min = A[1]
2 for i = 2 to length(A)
3 if min > A[i]
4 then min = A[i]
5 return min

n-1 comparisons.

Min and max together

• compare every two elements A[i], A[i+1]
• Compare larger against current max
• Smaller against current min
• $3\lceil n / 2 \rceil$

Selection in expected O(n)

Randomised-select(A, p, r, i)
if p=r then return A[p]
q = Randomised-Partition(A,p,r)
k = q – p + 1 // nr of elements in subarr
if i<= k
then return Randomised-Partition(A,p,q,i)
else return Randomised-Partition(A,q+1,r,i-k)

Conclusion

• Sorting in general $O(n \log n)$
• Quicksort is rather good
• Linear time sorting is achievable when one does not assume only direct comparisons
• Find $i'$ th value – expected $O(n)$
• Find $i'$ th value: worst case $O(n)$ – see CLRS

Ok...

• lists – a versatile data structure for various purposes
• Sorting – a typical algorithm (many ways)
• Which sorting methods for array/list?
• Array: most of the important (e.g. update) tasks seem to be $O(n)$, which is bad

Q: search for a value X in linked list?

A. $O(1)$
B. $O(\log n)$
C. $O(n)$
Can we search faster in linked lists?

- Why sort linked lists if search anyway O(n)?

- Linked lists:
  - what is the “mid-point” of any sublist?
  - Therefore, binary search can not be used...

- Or can it?

Skip lists

- Build several lists at different “skip” steps

  - O(n) list
  - Level 1: \( \sim n/2 \)
  - Level 2: \( \sim n/4 \)
  - ...
  - Level \( \log n \) \( \sim 2-3 \) elements...

Skip List

typedef struct nodeStructure *node;
typedef struct nodeStructure {
    keyType key;
    valueType value;
    node forward[1]; /* variable sized array of forward pointers */
};

What is a Skip List

- A skip list for a set \( S \) of distinct (key, element) items is a series of lists \( S_0, S_1, \ldots, S_h \) such that
  - Each list \( S_i \) contains the keys of \( S \) in nondecreasing order
  - Each list is a subsequence of the previous one, i.e., \( S_i \subseteq S_{i+1} \)
  - List \( S_h \) contains only the two special keys
- We show how to use a skip list to implement the dictionary ADT

Skip Lists

Fig. 1. A skip list with \( \alpha = 6 \) nodes and \( \log \alpha = 3 \) levels.
**Randomized Algorithms**

- A randomized algorithm performs coin tosses (i.e., uses random bits) to control its execution.
- It contains statements of the type
  
  ```java
  h ← random()
  if h = 1
    do A
  else
    do B
  ```
- The worst-case running time of a randomized algorithm is often large but has very low probability (e.g., it occurs when all the coin tosses give “heads”)
- We use a randomized algorithm to insert items into a skip list

**Search**

- We search for a key x in a skip list as follows:
  - We start at the first position of the top list
  - At the current position i, we compare x with y = key(aj[i])
    
    - if x = y: we return true
    - if x < y: we move to the current successor
    - if x > y: we move to the current predecessor
  - If the search proceeds past the bottom list, we return NO_SUCH_KEY

**Example:** search for 78

**Insertion**

- To insert an item (x, y) into a skip list, we use a randomized algorithm:
  - We repeatedly toss a coin until we get tails, and we denote with t the number of times the coin came up heads
  - If a head, we add the skip list new lists S0, S1, ..., St, each containing the two special keys
  - We search for x in the skip list and find the positions p0, p1, ..., pt of the items with largest key less than x in each list S0, S1, ..., St
  - For i = 0, ..., t, we insert item (x, y) into list Si after position pi

**Example:** insert (15, 5)

**Deletion**

- To remove an item with key x from a skip list, we proceed as follows:
  - We search for x in the skip list and find the positions p0, p1, ..., pt of the items with key x, where position pi is in list Si
  - We remove positions p0, p1, ..., pt from the lists S0, S1, ..., Si
  - We remove all but one list containing only the two special keys

**Example:** remove key 14

**Implementation v2**

- We can implement a skip list with quad-nodes
  - A quad-node stores:
    - item
    - link to the node before
    - link to the node after
    - link to the node below
    - link to the node above
  - Also, we define special keys PLUS_INF and MINUS_INF, and we modify the key comparator to handle them

**Example:**
Skip Lists

• The space used by a skip list depends on the random bits used by each invocation of the insertion algorithm.
• We use the following two basic probabilistic facts:
  Fact 1: The probability of getting i consecutive heads when flipping a coin is $\frac{1}{2^i}$.
  Fact 2: If each of n items is present in a set with probability p, the expected size of the set is $np$.
• The expected number of nodes used by the skip list is $\sum_{i=0}^{\infty} \frac{n}{2^i} = 2n$.
• Thus, the expected space usage of a skip list with n items is $O(n)$.

Search and Update Times

• The search time in a skip list is proportional to:
  – the number of drop-down steps, plus
  – the number of scan-forward steps.
• The drop-down steps are bounded by the height of the skip list and thus are $O(\log n)$ with high probability.
• To analyze the scan-forward steps, we use yet another probabilistic fact:
  Fact 4: The expected number of coin tosses required in order to get tails is 2.
• When we scan forward in a list, the destination key does not belong to a higher list.
  – A scan-forward step is associated with a former coin toss that gave tails.
  – By Fact 4, each list the expected number of scan-forward steps is 2.
  – Thus, the expected number of scan-forward steps is $O(\log n)$.
• We conclude that a search in a skip list takes $O(\log n)$ expected time.
• The analysis of insertion and deletion gives similar results.

Height

• The running time of the search an insertion algorithms is affected by the height h of the skip list.
• We show that with high probability, a skip list with n items has height $O(\log n)$.
• We use the following additional probabilistic fact:
  Fact 3: If each of n events has probability p, the probability that at least one event occurs is at most $np$.
• By picking $i = \log n$, we have that the probability that $X_{3\log n}$ has at least one item is at most $\frac{3\log n}{2^{3\log n}} = \frac{n}{2^n} = \frac{1}{e^3}$.
• Thus a skip list with n items has height at most $3\log n$ with probability at least $1 - \frac{1}{e^3}$.

Summary

• Using a more complex probabilistic analysis, one can show that these performance bounds also hold with high probability.
• Skip lists are fast and simple to implement in practice.

Conclusions

• Abstract data types hide implementations.
• Important is the functionality of the ADT.
• Data structures and algorithms determine the speed of the operations on data.
• Linear data structures provide good versatility.
• Sorting – a most typical need/algorithm.
• Sorting in $O(n \log n)$: Merge Sort, Quicksort.
• Solving Recurrences – means to analyse.
• Skip lists – $\log n$ randomised data structure.