vg101: Introduction to Computer Programming

RC 7

CHEN Xiwen

2019/7/5 (FRI)

Data Adaptability

`void*` type `[demoVoidPtr.c]`

- From dynamic memory allocation we know that

  1. The size of the memory are decided by programmers, not the computers.
  2. We can allocate memory for any kind of data.
  3. The returned value of `malloc` and `calloc` is `void*`.

- Void pointer? Pointer of "any type".

  ```c
  int a = 10;
  char b = 'a';

  void* p = &a;
  p = &b;
  ```

- Key points:

  1. Void pointers enables us to allocate memory for any kind of data.
  2. Void pointers in `C` are used to implement generic functions in C.

     e.g., `qsort` and `bsearch` functions in `<stdlib.h>`

  3. Void pointers cannot be dereferenced. The following does not compile.

     ```c
     #include <stdio.h>

     int main() {
         int a = 10;
         void* p = &a;
         printf("%d\n", *p);
         // printf("%d\n", *(int*)p); // works fine
         return 0;
     }
     ```

  4. Pointer arithmetics are not allowed for void pointers in the `C` standard. However, `gcc` treats the size of `void` as 1.

```c
1  #include <stdio.h>
2  int main() {
3      int c[5] = {1, 2, 3, 4, 5};
4      void* p = &c;
5      p = p + 2 * sizeof(int);
6      printf("%d\n", *(int*)p);
7      return 0;
8  }
```

Q: Can you give an explanation of 3 and 4?

Algorithms

- Divide & conquer

  Q: Can you think about a divide and conquer scheme to calculate two polynomials? For example, given $[5, 0, 10, 6]$ and $[1, 2, 4]$ which represent the following polynomials

  $$p_1(x) = 5 + 0 \times x + 10 \times x^2 + 6 \times x^3, \quad p_2(x) = 1 + 2 \times x^2 + 4 \times x^3$$

  we want to get the output

  $$p_1 p_2(x) = 5 + 10 \times x + 30 \times x^2 + 26 \times x^3 + 52 \times x^4 + 24 \times x^5.$$

  A simple algorithm is to use a nested loop to iterate every coefficient in the two polynomials, adding the multiplication of two coefficients according to the power of x and add to the current coefficient value. But can you think of a better algorithm using divide and conquer?

  A: Consider polynomials p_1 and p_2 as

  $$p_1 = p_1^{(0)} + p_1^{(1)} x^{n/2}, \qquad p_2 = p_2^{(0)} + p_2^{(1)} x^{n/2},$$

  where p_1 and p_2 are of order $n = 2^k$, p_is are polynomials of order $n/2$. Then

  $$p_1 p_2 = \left(p_1^{(0)} + p_1^{(1)} x^{n/2}\right)\left(p_2^{(0)} + p_2^{(1)} x^{n/2}\right) = p_1^{(0)} p_2^{(0)} + (p_1^{(0)} p_2^{(1)} + p_1^{(1)} p_2^{(0)}) x^{n/2} + p_1^{(1)} p_2^{(1)} x^n,$$

  where all the $p_i^{(0)}$ and $p_i^{(1)}$ are of polynomials with order $n/2$. Then using

  $$A = \left(p_1^{(0)} + p_1^{(1)}\right)\left(p_2^{(0)} + p_2^{(1)}\right), \qquad B = p_1^{(0)} p_2^{(0)}, \qquad C = p_1^{(1)} p_2^{(1)},$$

  we have

  $$p_1 p_2 = A + (A - B - C) x^{n/2} + C.$$

  The recurrence relation is given by

  $$T(n) = 3T\left(\frac{n}{2}\right) + O(n),$$

  where $T(n)$ is the cost of this algorithm when the polynomials are of order n. Modifications can be applied when n is not a power of 2, or the two polynomials have different orders.

- Randomized algorithms `[demoMC.c]`

  [Monte-Carlo methods] Simple example: how can you approximate π?

- Algorithmic examples

  1. Searching `[demoSearch.c]`
     - Linear search
     - Binary search

2. Sorting (of length n array in ascending order) `[demoSort.c, sort.h, sort.c]`

   The following pseudocode assumes starting from index 1.

   - Bubble sort

     Traverse the array for n times, for each traverse, we have additionally decided the largest element in the remaining array and put that element at the last of the remaining array.

     ```
     for i = 1, ..., n do
         for j = 1, ..., n - 1 do
             if arr[i] > arr[j] then
                 swap(arr[i], arr[j]);
             end
         end
     end
     ```

     Q: Can you recognize a little bit of improvement for the previous algorithm?

   - Insertion sort

     ```
     for i = 2, ..., n do
         insert arr[i] into the appropriate position in the sorted array arr[1], ..., arr[i - 1];
     end
     ```

   - Selection sort

     ```
     for i = 1, ..., n - 1 do
         find the smallest element in arr[i], ..., arr[n];
         swap the element with arr[i];
     end
     ```

   - Merge sort: split array into two subarrays of roughly equal sizes.

   - Quick sort:

     1. Choose a pivot
     2. Put elements less than the pivot to the left of the pivot
     3. Put elements larger than or equal to the pivot to the right of the pivot
     4. Sort the subarrays

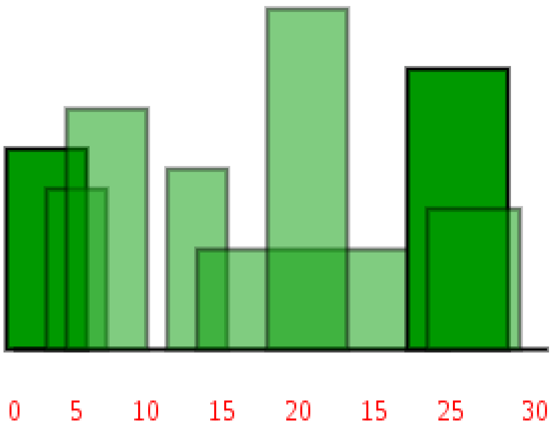
     ```
     quick_sort(arr, left, right)
         if left >= right do
             return;
         end
         choose an array element as pivot with position p_ind;
         partition the array into two subarrays using pivot;
         quick_sort(arr, left, p_ind - 1);
         quick_sort(arr, p_ind + 1, right);
     end
     ```

Practices

GeeksforGeeks

- Divide and conquer
    1. The Skyline problem. `[skyline.c]`

       Given n rectangular buildings in a 2-dimensional city, compute the skyline of these buildings, eliminating hidden lines.

       Input: an array of buildings, where a building is represented as an array `{left, height, right}`.

       Output: an array of rectangular strips as skyline, where a strip is represented as an array `{left, height}` and the `left` boundaries of the strip follow an ascending order. The rectangular strips give the overall shape of the buildings.

       > Input:
       >
       > {(1, 11, 5), (2, 6, 7), (3, 13, 9), (12, 7, 16), (14, 3, 25), (19, 18, 22), (23, 13, 29), (24, 4, 28)}
       >
       > Output:
       >
       > {(1, 11), (3, 13), (9, 0), (12, 7), (16, 3), (19, 18), (22, 3), (25, 0)}

  Divide and conquer scheme: divide the buildings into two parts and merge the skylines of the two parts.

- Modify the binary search algorithm so that it is randomized. `[randomized.c]`